UNIVERSITY OF OSLO
Department of Informatics

Using virtual machines for integrity checking

Master thesis

Mads Bergdal

Trond Arne Sørby

May 1st 2007

Abstract

Today's arms race between the attackers and defenders of computer systems seems like a never ending story. Traditionally, the battle has been fought outside the computer's operating system kernel, but in recent years the advent of kernel level malware has moved the battlefield inside the operating system, thus incapacitating many of the before trusted security mechanisms. When this happens the operating system can no longer be trusted, and new kinds of security tools must be developed.

This thesis looks at the potential of virtualization as a platform for performing integrity checking of a running operating system's kernel. In theory, the use of virtualization should make it possible to establish a platform of trust in the system, even when the kernel of a virtualized guest kernel has been subverted.

The idea of monitoring an attacked system from a different protection domain than the attacked system is not new. The use of virtualization brings some extra benefits though: High visibility to the monitored system *and* good protection from outside attackers. Traditional computer surveillance systems have been forced to compromise between these two properties.

The reader is in this thesis introduced to the concept of kernel level malware, virtualization techniques and the internals of the Linux kernel. An architecture designed to address some of the problems surrounding the integrity checking of a running kernel, is presented. The details of this architecture is discussed, and a working prototype putting the architecture to the test against a suite of real attacks, is constructed.

Preface

This report presents the results of our master thesis "Using virtual machines for integrity checking". It is written as part of our Master degrees in Computer Science at the University of Oslo. The work presented is based on a problem presented by security community at the Norwegian Defence Research Establishment (FFI). The problem statement was given to us by our daily supervisor at FFI, Ane Daae Weng. In addition to having Weng as our daily supervisor, professor Chunming Rong from the University of Oslo has been our main supervisor.

The work with this thesis has opened our eyes to many new and exiting topics within the area of computer security. Although the work with this thesis has been challenging, it has been immensely educating. Our knowledge of virtualization and kernel level malware was as best superficial at the beginning of this work. Our C and Python programming skills have also been thoroughly tested. In general our interest in the field of computer security has increased dramatically, and we will both continue down this path in our professional careers.

We would first use this opportunity to thank our families for putting up with us during the work on this thesis. We would also like to thank our two supervisors Ane Daae Weng and Chunming Rong for their valuable inputs along the way. A special thank also to Ronny Windvik, project manager at FFI, for his inputs on our work. We would also like to thank Bryan D. Payne of the XenAccess project. His innovative work has helped us a lot with the implementation of our prototype. At last we also feel that we owe the XenSource community in general a special thank. The Xen virtualization technology is a complex piece of software, and their input and guidance have been very helpful to us.

Oslo, May 1st 2007

Trond Arne Sørby Mads Bergdal

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The topic for this master thesis is the verification of the integrity of an operating system's running kernel, focusing on the potential that virtualization technology offers as a platform for integrity analysis.

Virtualization provides some highly desirable properties as a security platform, as examplified by the recent research efforts using virtualization as a building block for constructing intrusion detection systems (IDS) [37] [48]. Traditional IDS have had to choose between the high attack resistance of a network-based IDS (NIDS) or the high amount of machine state visible to a host-based IDS (HIDS). The use of virtualization has made it possible to get the best from both these approaches: a high attack resistance and the ability to directly inspect the hardware state of a monitored host.

In this thesis, virtualization is utilized for the purpose of analysing the run-time integrity of the operating system's kernel. Both a general architecture and a working prototype of such a system are presented.

Section 1.1 gives a brief background of the motivations underlying the thesis. In section 1.2, the thesis' theme will be given a concrete representation in the form of a problem definition. The methodology used in the thesis is described in section 1.3. A structural view of the thesis is provided in the final section of this chapter.

1.1 Motivation

Millions of users worldwide rely on the correctness and security of their computer systems. The networked world following the global adoption of the internet provides a tempting playground for people with malicious intent. Their motivation may vary from the thrill of the challenge to the prospect of finanical gains.

The complexity of the general purpose operating systems of today, makes verifying their security a daunting task. In fact, only the most specialized systems seek to offer any kind of security guarantees [3], making the need constant patching of security holes in the major operating systems seem natural.

While the ultimate goal is to prevent the system from being compromised, the reality is that even the detection of a compromise is as big a challenge. The possibility of the compromise staying undetected, raises serious questions about the trust that can be placed in the system.

The kernel of an operating system provides the foundation for all the other software running on the system including software tasked with ensuring the security of the system, such as virus scanners, personal firewalls and host-based intrusion detection systems. Because the detection mechanisms typically rely on the functionality and data provided by the kernel, a compromise of the kernel can aid the attackers in the deception of detection mechanisms by filtering their view of the system. As a result, all data originating from a compromised kernel must be assumed to be deceptive.

The ability to maintain undetected access to a compromised machine, has been the motivation behind the development of much of today's kernel-level malware. As a result, recent years have seen increasingly sophisticated attacks directed at the kernel. The amount of malware operating fully in kernel mode is steadily increasing, making kernel-level malware gaining the attention of not only security researchers, but of the computer industry at large [46]. Protecting the kernel seems to be a increasingly important concern for operating system vendors, as exampli-fied by the PatchGuard mechanism found in Windows Vista, aimed at protecting vital kernel structures and code from unauthorized modification [24].

1.2 Problem statement

This thesis can be seen as a continuation of the work done by Tobias Melcher [54]. While Melcher provided a general model for integrity checking using virtualization, the implementation of that model was limited to the integrity checking of

files.

While the work of Melcher provided a starting point, the model needed to be refined and improved, with a bigger emphasis placed on the in-memory representation of the kernel and its data structures. It was concluded that a further exploration of the subject would necessitate the implementation of prototype capable of performing integrity checks of the kernel's memory space.

The problem statement given by our daily supervisor was thus very much centered around the implementation of a proof-of-concept integrity analyser:

Integrity checking of a running kernel using virtualization. *What are the possibilites and limitations in using virtualization as a platform for the integrity checking of a running kernel? Given access to the in-memory representation of the kernel, can its integrity be verified? The result of the work should include, in addition to a written report, an implementation of a proof-of-concept tool for integrity analysis.*

1.2.1 Research questions

Based on the above problem statement, the main question of this thesis can be formulated:

How can virtualization aid in the verification of the integrity of the memory of an operating system's kernel?

From the main question, two subquestions can be identified:

1. *How can virtualization provide a platform for integrity analysis?* The answer must address both the functional requirements and the security implications.

2. *Can the integrity of an operating system's kernel be verified?* An answer will require a discussion of the integrity property and the preconditions required for the verification of integrity.

1.3 Research methodology

The work conducted can be divided into four phases, a preliminary phase, a phase of designing and implementing the prototype, an experimental phase and finally, a phase for discussion and evaluation.

1.3.1 Preliminary phase - literature study

The purpose of the preliminary phase was to gain an understanding of the problem area.

A comprehensive literature study on the subjects of virtualization, operating systems and kernel-level malware was conducted. The prior work of Melcher, and the background material used therein, gave a starting point for the exploration of the problem area.

1.3.2 Design and implementation

Following the litterature study, the design and implementation stage began. In this phase, a model for virtualization-based integrity analysis was formulated based largely on existing works. The requirements for an implementation was formulated, and the technologies used in the implementation were decided upon.

The prototype is exposed to the internals of both the virtualization technology and the operating system to be monitored, making the analysis of their implementation an integral part of the development of our system. Due to sparse documentation of large parts of the virtualization technology, thorough code reviews were needed for understanding its inner workings.

1.3.3 Experimenting

While the implementation phase focused on the construction of mechanisms, this phase was all about putting the mechanisms to work. Integrity policies were created and put to the test, and the experiences gained formed a basis for the later discussion of the potential and limitations of the system.

The creation of tests to be performed, required both a thorough understanding of the technical aspects of the system as well as a knowledge of the techniques used in malware targetting the kernel.

1.3.4 Discussion and evaluation

In this phase the potential and limitation offered by the use of virtualization as a platform for integrity analysis were discussed. Furthermore, the problems of integrity analysis in general, and the approach taken by this thesis, was compared with the approaches taken in other research. The discussions based themselves on available research and the experiences gained from using the prototype.

1.4 The scope of the thesis

The focus of this work is on the Linux operating system with Xen as the virtualization platform, running on the Intel x86 architecture. Still, the goal has been to maintain as much generality as possible, in both the model and solutions developed.

File integrity checking falls outside the scope of the thesis, the focus is on verifying the integrity of the kernel as it appears *after* being loaded into memory. For a demonstration and discussion of the use of virtualization for integrity checking of files important to the kernel, interested readers are referred to the work of Melcher [54].

Attackers are assumed to having already obtained unrestricted access to the administrative account of the system. The ways in which such access can been obtained are not a topic for this thesis.

1.5 Summary of results

The results obtained in this thesis suggest that virtualization provides a near ideal environment for performing integrity analysis of the kernel. The visibility of the machine state of the monitored host combined with the isolation guarantees provided by virtualization raises the trustworthiness of the data obtained, while keeping the analysing mechanism out of harms way. However, the platform is not without potential attack surfaces, and an effort is made identifying those.

The thesis also shows that the verification of integrity is problematic, even when the trustworthiness of the analyser and the correctness and completeness of the underlying data set can be assumed.

1.6 Thesis outline

The thesis is comprised of the following chapters:

Chapter 1 provides the motivation for this thesis, defines the problem area and states the research questions.

Chapter 2 gives a description of the aspect of computer security known as *integrity*, and discusses the *mechanisms* enforcing the *integrity policies* of a system.

Chapter 3 gives an overview of the Linux kernel, the operating system whose kernel will be the subject of integrity analysis.

Chapter 4 is a survey of the threats to kernel integrity, focusing on the kernel-level malware known as kernel mode rootkits.

Chapter 5 serves as an introduction to the topic of virtualization, establishing a terminology and giving several examples of virtualization solutions currently in use.

Chapter 6 takes a closer look at Xen, the virtualization solution used in the implementation.

Chapter 7 , after identifying the necessary requirements, provides a general architecture for an integrity analysis framework.

Chapter 8 describes the proposed prototype of an actual implementation of the architecture presented in the preceeding chapter.

Chapter 9 shows how the prototype was put to use in implementing integrity checks of selected parts of the kernel.

Chapter 10 describes how the system was tested using kernel-malware attacks.

Chapter 11 discusses the work done, placing it in a wider context of integrity-related research and evaluating the approaches and techniques used for this thesis.

Chapter 12 provides the conclusions of the thesis, giving some suggestions for future work.

Appendices A-E contain an introduction to loadable kernel modules, a description of the object types of the VFS, instructions for the installation and usage of Xen, examples of kernel-level malware and, finally, the source code for the Chili framework.

Part I

Background

Chapter 2

Integrity of computer systems

Integrity is a key aspect of computer security, and ensuring system integrity continues to be a significant challenge for the system administrators of the world.

The goal of this chapter is to serve as an introduction to the topic of integrity, with focus on the attacking and protection of integrity. Recent trends of integrity attacks will be identified, establishing the kernel of the operating system as an arena for the battle over integrity. A brief survey of current research concerning integrity will be given, with an emphasis on the topics motivating the work of this thesis.

In section 2.1, a formal definition of integrity together with its related terms will be given. Section 2.2 gives an overview of the threats to system integrity, with a focus on the most common attacks and the countermeasures developed through the years. Section 2.3 describes the current trends of integrity attacks, and the efforts made by both research and industry to meet the challenges imposed on them. Concluding the chapter, section 2.4 summarizes some of the research work already done in this area of computer security.

2.1 Definition of terms

As this thesis deals with the integrity property of the operating system, a clarification and definition of integrity and its related terms are needed. The definitions in this sections are taken from the textbook "Computer Security" by Matt Bishop [31].

Bishop provides a formal definition of integrity:

> Let X be a set of entities and let I be some information or a resource. Then I has the property of *integrity* with respect to X if all members of X trust I.

Bishop identifies several aspects of integrity: *Data integrity*, relating to the content of the information, and *origin integrity*, relating to the the source of the data, also known as *authentication*. When integrity relates to a resource rather than information, integrity means that the resource functions correctly. This aspect of integrity is often called *assurance*.

The rules stating what is, and what is not, allowed for the integrity to be preserved, are called *integrity policies*. They describe not only *how* information or resources may be altered, but also by *whom*.

The mechanisms tasked with enforcing integrity policies are usually divided into two classes: *prevention* mechanisms and *detection* mechanisms. While prevention mechanisms try to keep integrity violations from occurring in the first place, detection mechanisms are limited to reporting that an integrity violation has indeed occurred.

The definition of integrity given by Bishop refers to the trustworthiness of data or resources, making the question of *system integrity* a question of the trust that can be placed in the system. A critical observation made by Bishop, is that the effectiveness of any security mechanism depends on the trust that can be placed both in the underlying base on which the mechanism is implemented and the correctness of the implementation.

The notion of the integrity of higher layers being dependent on the integrity of lower layers, has been stated even more explicitly by Arbaugh et al. [29]. They state that, presuming the validity of the hardware layer (the lowest layer), integrity of a layer can be guaranteed if and only if: (1) the integrity of the lower layers is checked, and (2) transitions to higher layers occur only after integrity checks on them are complete. By inductively building on the integrity of lower layers, system integrity can be guaranteed.

2.2 The fight for integrity

In this thesis, the term *attack* will be used for all acts threatening integrity, be they accidental or deliberate. The entity carrying out the attack, whether a person or fate itself, will be called the *attacker*. The opposite terms, *defence* and *defender*, will be used when the intent is to either *prevent* or *detect* the compromise of integrity.

After an attack is detected, the extent of the integrity breach can be difficult to establish. Note that not all attacks necessarily result in integrity compromises. It is, however, difficult to prove the integrity of a system on which a succesful attack has been launched. If an attack is only detected but not prevented, it is common practice to treat the integrity of the system as compromised [72].

2.2.1 Attacks

Using the above definitions, all actions resulting in the trustworthiness of the system being questioned can be characterized as integrity attacks.

The goal of the attackers is often not limited to obtaining access to systems, but also maintaining that access undetected. Attacks range from the completely non-technical forms of social engineering to the highly technical forms as the exploitation of vulnerabilities in code.

Malware

Malware has emerged as the term for describing software with a malicious intent, including viruses, worms, trojan horses and rootkits.

Vulnerabilites

Vulnerabilities stem from bugs or flaws in the design, implementation, configuration or use of software. An attacker can exploit vulnerabilities to gain access or to escalate existing privileges on a system. Often, an attack consists of gaining access by exploiting known vulnerabilities, and ensuring future access by installing malware specifically designed for that task.

2.2.2 Defences

Mechanisms for dealing with the integrity of computer systems can be divided into two groups: *prevention* mechanisms and *detection* mechanisms.

2.2.3 Detection

Detection software can usually be divided into various form of malware detectors and systems for detecting other forms of intrusions, commonly referred to as intrusion detection systems (IDS).

Detection software employs a multitude of techniques:

- *Signature detection*
  Signature detection is perhaps most widely known from anti-virus software,

but the technique can also be used for other kinds of malware detectors or as part of an IDS. The assumption is that most malware or exploits contain specific patterns or code sequences that can be used to identify them. Signatures can be very effective against known attacks, but the technique has definitive weaknesses. It requires the frequent updating of a database of signatures, and it protects only against known attacks.

- *Anomality detection*
  Anomality detection makes the assumption that it is possible to state what is the expected behavior of the system. Any unexpected behaviour is seen as suspicious, and may be evidence of an intrusion.

- *File integrity checkers*
  File integrity checkers are used for detecting changes made to key system binaries, configuration files and other files that are not expected to change during normal usage. Typically, the integrity checker maintains a database of cryptographic checksums of the file contents. The integrity checker computes new checksums at (ir)regular intervals, comparing them to the ones stored in the database. Changes made to monitored files will result in a different checksum, causing the file integrity checker to issue an alert.

- *Auditing*
  Auditing is the analysis of a system's log records. By keeping extensive logs of noteworthy events in the system, violations of security policies may be detected and the sequence of events leading up to the breach may be examined.

2.2.4 Prevention

While detection mechanisms merely report when the integrity of a system is compromised, other mechanisms try to hinder the system from being compromised in the first place.

Over the years, a set of *best practices* for securing a system has emerged, and the process of employing these practices are often referred to as *hardening* the system [72] [16]. This will typically include turning off non-essential services, running processes with only those privileges needed for completing their tasks, enforcing a strict password policy and keeping the system patched with security updates.

Note that while a hardened system offers a harder challenge for an attacker, no guarantees can be given on the security of the system.

2.3 Trends

At the end of section 2.1, it was noted how the integrity of a layer depends on the integrity of its lower layers. This key insight can be seen as causing the trend of attacking increasingly lower layers of the software stack.

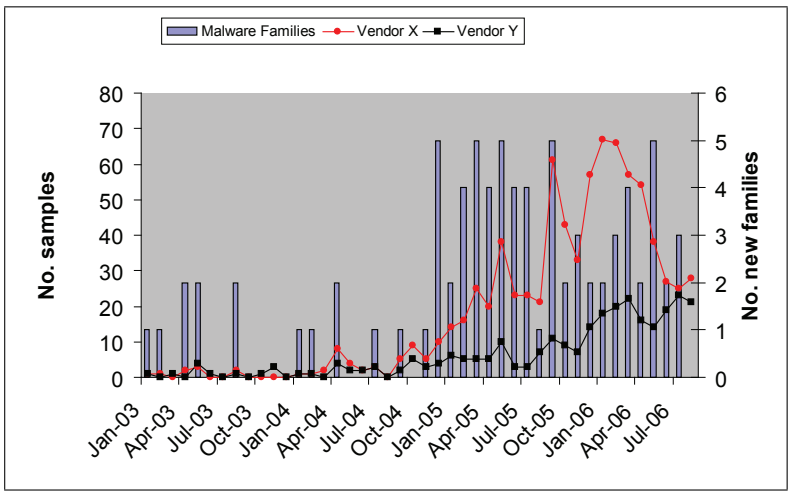

Figure 2.1: Number of malicious kernel malware samples found per month by two anitvirus vendors, together with the number of new malware families found to include kernel-mode components [46].

Malware operating within the kernel has seen a steadily increase in popularity during the last few years. Figure 2.1, taken from Kasslin [46], shows the increasing growth of kernel-level malware for the Windows operating system. Kasslin explains the rise in popularity by the increased motivation for malware authors to keep their malware hidden from detection.

Detection and prevention mechanisms placed within the system being monitored all rely on the view of the computer system presented to them by the operating system. By making malicious modifications to the operating system kernel, the information requested by these mechanims may be filtered by the attacker. As an example, recent years have seen the introduction of kernel-level malware, capable of keeping the presence of both files and processes hidden from the detection mechanisms.

In chapter 4, both kernel-level malware and the motivations for their existence will be given a more thorough look. For now, it is sufficent to note that today, the battle over system integrity is in large parts being fought inside the kernel of the operating system.

2.4 Research efforts

A significant amount of research relating to integrity exists. In this section, some research topics found to be of interest are presented.

2.4.1 Code attestation

Seshadri et al. [70] defines code attestation as a process where a trusted entity, known as the *verifier*, verifies the software stack that runs on another entity, called the *attestation platform*. Integrity measurements of the attestation platform's software stack are taken by a *measurement agent* running on the attestation platform, and sent to the verifier. The integrity measurements taken from the attestation platform enable the verifier to detect modifications in the software stack of the attestation platform.

The Trusted Computing Group (TCG) has released a set of standards describing how a separate tamper-resistant coprossesor, known as the Trusted Platform Module (TPM), can be used for taking integrity measurements of a system [21]. The measurements consist of computing a checksum of the code that is to be loaded, using a hash function. While the primary motivation for the development of the TPM was to provide an immutable base capable of ensuring the secure bootstrapping of a machine [29], the mechanism can be used for providing integrity statements of other parts of the software stack [68]. Seshadri et al. [70] describes how the integrity measurements taken by the TPM can be used for guaranteeing *load-time code attestation*, whereby the verifier obtains a guarantee of what code was loaded into the system memory initially.

The guarantees obtained from load-time code attestation are limited to what code is loaded into memory, hence making the code vulnerable from in-memory patching after it is loaded. An example of *run-time code attestation* is Copilot [57], a kernel integrity monitor running on a PCI add-in card. Copilot performs integrity measurements of the kernel memory at regular intervals, thus detecting changes made after the loading of the kernel code.

Another example of code attestation is the work on Pioneer, which unlike the other examples does not require any hardware extensions added to the attestation platform [70]. Pioneer establishes a trusted computing base on the attestation platform, called the *dynamic root of trust*. The dynamic root of trust is created using a *verification function*, a *self-checking* function that computes a checksum of its own instructions. The checksum function is constructed in such a way that any tampering will result in either a wrong checksum or a noticeably slowdown of the

computation. Using a challenge-response protocol, it is shown that if the verifier receives the correct response within a certain amount of time, it can be guaranteed that the verification function code on the attestation platform has executed unmodified.

2.4.2 Virtualization

Using virtual machines as a security mechanism is nothing new. Bishop describes how virtual machines can be used for isolation purposes, preventing the processes of the virtual machine from accessing the underlying computer system [31].

Garfinkel and Rosenblum show how virtualization, in addition to the isolation it provides, offers the possibility to directly inspect the hardware state of the virtual machine [37], thus providing a high degree of visibility. In their view, the combination of both visibility and isolation makes virtualization especially suitable as a platform for intrusion detection systems (IDS). The two dominant IDS architectures of today, network-based intrusion detection systems (NIDS) and host-based intrusion detection system (HIDS), each offers one at the cost of the other. A NIDS offers a high degree of isolation, but poor visibility, while a HIDS provides good visibility at the cost of isolation.

Chapter 3

The Linux operating system

3.1 Introduction

The Linux operating system is the result of the work started by Linus Torvalds in 1991. Linux has been chosen as the operating system of choice in this thesis mostly because its source code is open for everyone to review. This means that one is free to delve into the inner workings of the operating system to find out exactly how everything works. This chapter gives the reader a brief introduction to the Linux operating system. The focus will be on the parts of the Linux kernel that the reader will need to have a basic understanding of to understand the difficulties of integrity checking this complex piece of software.

A modern computer consists of a great deal of complex subsystems. Some of these are: one or more processors, memory, hard drives, keyboard, monitors, network interfaces and other input/output devices. It would be very difficult to make use of a computer by letting each piece of software written manage all of these resources by themselves. To keep all of these things manageable, computers are normally equipped with a layer of abstraction between the hardware and the software running on the computer [75]. This layer of software is the operating system. Figure 3.1 shows how a computer system can be viewed as a layered architecture with the operating system as the layer which all other programs rely on.

In short the operating system performs two main tasks: extending the machine and managing the machine's resources [75]. Firstly, the operating system extends the machine in the sense that a programmer doesn't have to worry about all the details of how hardware interact. A fundamental operation like reading from a

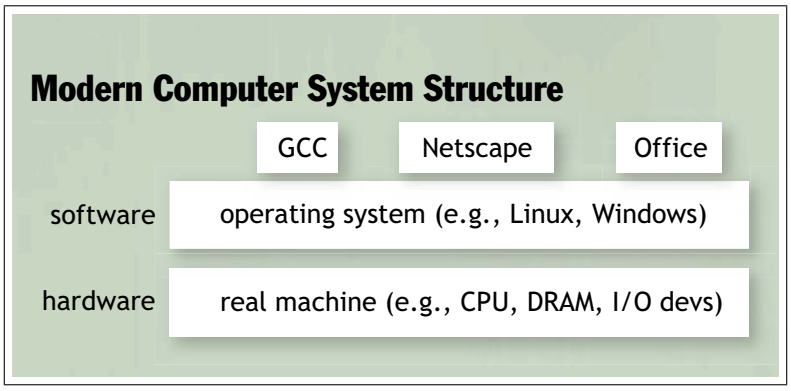

Figure 3.1: Computer system structure [63]

disk drive may sound like a straight forward operation, but it is actually quite complex, considering all the details of setting and reading device registers, managing read/write buffers, starting and stopping the device's motor and so on. All these details are hidden from the programmer by the operating system, making the use of various hardware on a system easy through well defined interfaces [75].

Secondly, the operating system helps with the management of the different resources in the system. A computer has many pieces of equipment which must all work seamlessly together. The operating system keeps programs and users separated, and make sure that all resource sharing is done in a proper and fair way.

The next section will give a short history of the origins of the Linux operating system. Then in section 3.3, some of the most important subsystems of the Linux kernel will be explained. The last section describes the Virtual File System (VFS), an important component of the Linux kernel. A knowledge of the different subsystems of the kernel is required for understanding how malicious code can subvert the kernel. Malicious kernel code will be the topic for the next chapter.

3.2 Background

In 1983, Richard Stallman started the GNU Project with the goal of creating a UNIX-like, POSIX-compatible operating system composed entirely of free software [11]. Some years later, in 1987, Andrew S. Tanenbaum (professor of Computer Science at the Vrilje University, Amsterdam) released an educational operating system named Minix. This operating system was (and still is) intended to be used for educational purposes. Therefore the Minix operating system was intentionally kept small and featureless. Minix was at the time the closest ting one could

get to a completely free clone of the commercial UNIX operating system [75].

A Finnish student, Linus Torvalds, decided to write another clone of the UNIX system which would eventually be a full-blown free production system. This system borrowed many ideas from the Minix system, and was first released in 1991. This was the start of the Linux operating system. It has now been ported to a wide range of hardware platforms, and a large number of Linux distributions exist, which bundle various other open source software with the Linux kernel [11]. The distribution used in this thesis is Debian GNU/Linux, one of the most feature rich distributions around today. Debian is also used as the base for many other Linux distributions [5].

3.3 The Linux Kernel

The Linux kernel is a monolithic kernel, meaning that all the device drivers and kernel extensions run in the most privileged level of the processor as a single program. This privileged level is also known as *kernel mode*. The x86[1] processor family offer four different execution states or protection levels [32], but most operating systems only use the innermost, kernel mode, and the outermost, *user mode*, protection levels.

3.3.1 Kernel modules

Kernel modules (or loadable kernel modules) are used in the Linux kernel to make the kernel able to provide support for new hardware, filesystems etc without having to build kernels with all this functionality precompiled into the kernel image. Without loadable kernel modules, the kernel binary would get much larger and more static. Adding new functionality to the kernel would thus require the kernel to be recompiled and the system to be rebooted.

Because kernel modules can be linked on demand, the kernel does not have to be bloated with hundreds of seldom-used programs. Nearly all higher-level components of the Linux kernel: filesystems, device drivers, executable formats and network layers can be compiled as modules.

Loadable kernel modules can also be used to subvert the kernel in various ways, as will be shown in chapter 4. Since the modules are linked into the kernel image, they get unrestricted access to all of the kernel's internal data structures and are therefore able to do unlimited damage if programmed to do so. Because of this,

[1] In this paper, CPUs belonging to the i386 family will be referred to as x86 processors.

many intrusion detection systems rely on monitoring the insertion of kernel modules [33]. More on how kernel modules are built and inserted into the kernel can be found in appendix A.

3.3.2 Memory management

Memory management is one of the most crucial tasks of the kernel. The kernel's memory manager is responsible for efficiently dividing the available physical memory to processes and threads and for freeing memory when it is no longer needed [75].

When talking about memory in the Linux kernel, it is important to distinguish between to main types of memory addresses: Virtual addresses and physical addresses [32]. The virtual address space is divided into *pages* (normally of 4kB each), and each page is contained in *page frames* in the physical memory.

3.3.3 Virtual memory

The basic idea behind virtual memory is that the combined size of the program, data and stack may exceed the amount of physical memory available for a process. This means that the set of memory references a process uses is different from the physical memory addresses actually accessed. It is the responsibility of a hardware circuit called the Memory Management Unit (MMU) to transform virtual addresses into physical ones.

Paging

The virtual memory subsystem uses a method called paging to manage the distinction between physical page frames and virtual pages. The *paging unit* inside the MMU thinks of all Random Access Memory (RAM) as partitioned into fixed-length *page frames*. Each page frame contains a *page*; that is, the length of a page frame coincides with that of a page. When the physical memory is insufficient to hold all the required pages, the MMU moves pages from primary memory (RAM) to secondary memory (hard disk) and marks the pages in the page table as swapped out to disk. This way the MMU frees up physical memory. The swapped page frame may later be retrieved from disk by the MMU when it gets a request for an address that corresponds to that particular chunk of memory. Such a request; a request for a page that is currently not in main memory, is called a *page fault*.

Memory layout

Figure 3.2 [32] shows how the first 3 MB of RAM are filled by Linux (assuming that the kernel requires less than 3 MB of RAM). Linux reserves some page frames

exclusively for the kernel code and data structures. The remaining parts of RAM are called *dynamic memory*, and are used by the user processes as well as by the kernel itself. The kernel uses dynamic memory for dynamically allocated kernel data structures, device driver buffers, code of a kernel module etc. Figure 3.3 [32] shows the dynamic memory areas.

The pages that are allocated for the kernel, are never swapped to disk [32]. This is an important observation, as the integrity checking system presented in part II of this thesis depends on being able to access all kernel memory at all times.

In figure 3.2 the symbol `_text`, corresponding to the physical address 0x00100000, denotes the address of the first byte of kernel code. The end of the kernel code is similarly identified by the symbol `_etext`. Kernel data is divided into two groups: initialized and uninitialized. The initialized data starts right after `_etext` and ends at `_edata`. The uninitialized data follows and ends up at `_end`.

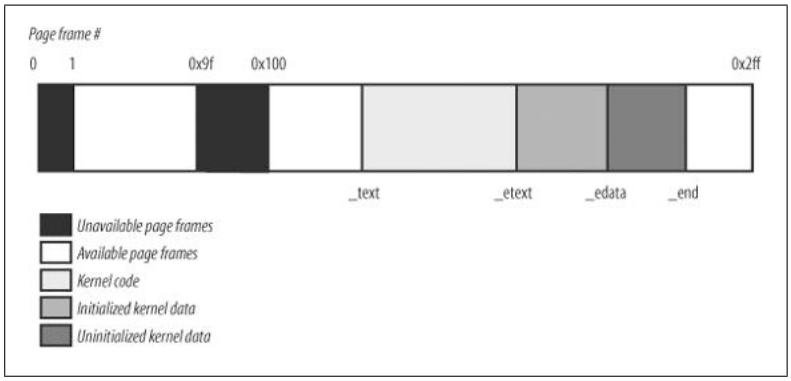

Figure 3.2: Kernel memory layout
[32]

The symbols appearing in figures 3.2 and 3.3 are not defined in Linux source code. They are produced while compiling the kernel, and the addresses of these symbols can be found in the file `System.map`, which is created as part of the kernel compilation process.

3.3.4 Context switch

In user mode the executing program cannot directly access the kernel data structures or the kernel programs. When in kernel mode, these restrictions do no longer apply. The processor provides special instructions for moving from user mode to kernel mode and vice versa. This switch between kernel and user mode is called a *context switch*. The services provided by the kernel are made available through

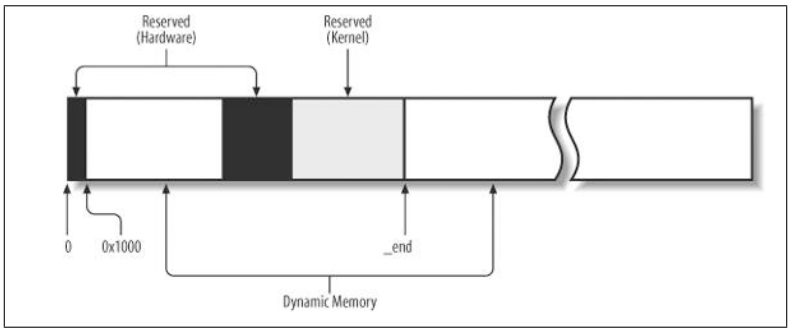

Figure 3.3: Dynamic memory
[32]

system calls (see section 3.3.6). Besides system calls, the kernel routines can be activated when the running process signals an exception, which the kernel then handles. A peripheral device may also issue an interrupt to signal the CPU of an event that requires attention. All such interrupts is handled by the interrupt handler within the kernel (see section 3.3.5).

3.3.5 Interrupt handler

An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor [32]. The Intel 80386 manual [45] talks of two types of interrupts: interrupts and exceptions, which are synchronous and asynchronous, respectively.

A system table called the *Interrupt Descriptor Table* (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler [32]. One such vector, 0x80 or 128 in decimal, will be given a detailed description in the next section. The fact that the addresses in the IDT decides what functions within the kernel should be called when an interrupt occurs, makes this an interesting point of attack for malicious code that have gained access to the kernel.

The *idtr* CPU register allows the IDT to be located anywhere in memory, and it specifies both the IDT base physical address and its limit (maximum length). The IDT must be initialized before enabling interrupts by using the `lidt` assembly language instruction.

3.3.6 System calls

Unix systems include several libraries of functions that provide APIs to programmers. Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should use [32]. These wrapper routines' only purpose is to issue the system calls.

When a user mode process invokes a system call, the CPU switches to kernel mode and starts the execution of a kernel function. A Linux system call can be invoked in two different ways (as will be discussed shortly). The result of both methods, however, is a jump to an assembly language function called the *system call handler*. To associate each system call number with its corresponding service routine, the kernel uses a system call dispatch table, which is stored in the *sys_call_table* array (see figure 3.4).

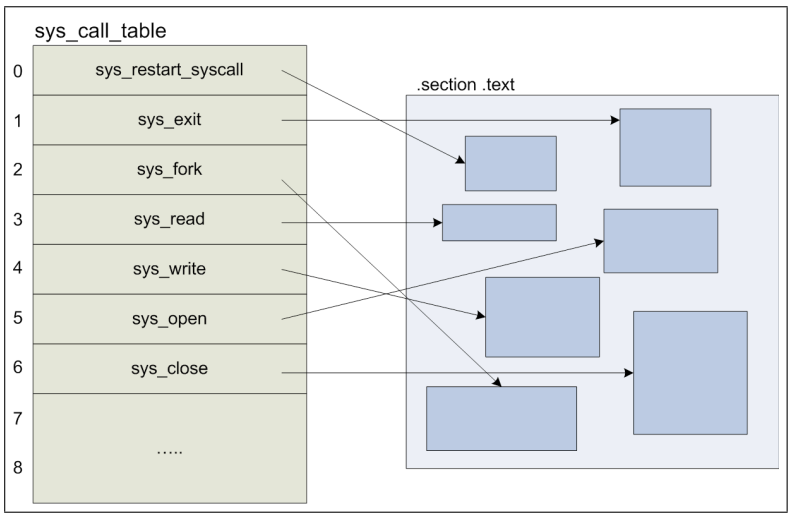

Figure 3.4: Syscall table

The system call handler is responsible for performing the following operations:

- Saving the contents of most registers in the Kernel mode stack.
- Handling the system call by invoking a corresponding C function. These functions are called *system call service routines*.
- Exiting from the handler: The registers are loaded with the values saved in the Kernel mode stack, and the CPU is switched back from Kernel mode to User mode.

Figure 3.5 [32] illustrates the relationships between the application program that invokes a system call, the corresponding wrapper routine, the system call handler and the system call service routine. The arrows denote the execution flow between the functions. The terms "SYSCALL" and "SYSEXIT" are placeholders for the actual assembly language instructions that switch the CPU from user mode to kernel mode and from kernel mode to user mode.

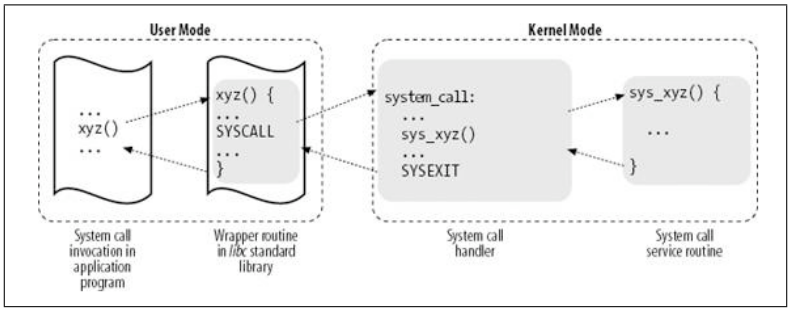

Figure 3.5: Invoking a system call
[32]

As mentioned above, a system call can be invoked in two different ways: By executing the `int 0x80` assembly language instruction or by executing the `sysenter` assembly language instruction. Below is a short description of these two methods.

int 0x80 The vector 0x80 (128) in the IDT points to the `system_call` function which is the service handler routine for all system calls entering via the int 0x80 instruction. The `system_call` function does a lot of work (permission checking, checking if the debugger is enabled etc.), but then eventually ends up calling the requested service routine by issuing the assembly instruction:

```
call *sys_call_table(0, \%eax, 4)
```

where the requested system call number is found in the eax register of the CPU [32].

sysenter The `int` assembly language instruction is inherently slow because it performs several consistency and security checks [32]. The `sysenter` instruction, also called "Fast System Call," provides a faster way to switch from user mode to kernel mode [32]. The wrapper routine in the standard library loads the system call number into the eax register and calls the

`__kernel_vsyscall` function. This causes the CPU to switch from user mode to kernel mode, and the kernel starts executing the `sysenter_entry` function. This again invokes the system call handler by executing a sequence of instructions identical to that starting at the `system_call` as described for `int 0x80` above.

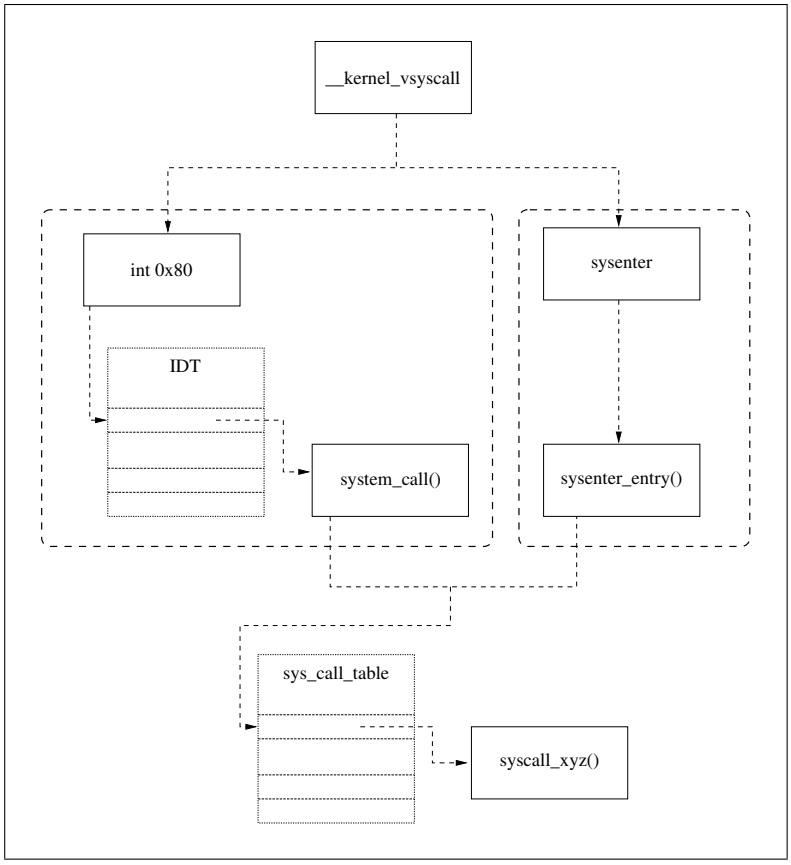

Figure 3.6: The system call chain

Being the link between user processes and kernel structures the system call chain is a very tempting target for malicious code. The whole process of invoking system calls can be (and in fact is) attacked in many ways by malicious kernel code, and will be discussed in more details in chapter 4. Figure 3.6 depicts the whole system call process.

3.4 The virtual file system

Linux supports a wide variety of file systems, and new ones can easily be added. Differences between file systems are made transparent to user space programs by the use of an interface layer known as the Virtual File System (VFS). The idea behind the VFS is to put a wide range of information in the kernel to represent the general features and behaviour of the different types of file systems [32]. The VFS thus defines the basic conceptual interfaces and data structures that all file systems support, while the implementation details are hidden in the code of the actual file systems. To the VFS layer and the rest of the kernel, each file system looks the same [51].

Because of the VFS, nothing in the kernel needs to understand the underlying details of the file systems, except the file systems themselves. Figure 3.7 illustrates how VFS abstracts away the specific file system details and provides user space with a generic file writing method. A user space program calls the write method of the system's standard library (for example, the C standard library, libc), which in turn calls the generic `sys_write()` system call. The system call then determines the actual file writing method to be used and invokes it, causing the data to be written to physical media. On one side of the system call is the generic VFS interface, providing the frontend to user space; on the other side of the system call is the file system-specific backend, dealing with the implementation details [51].

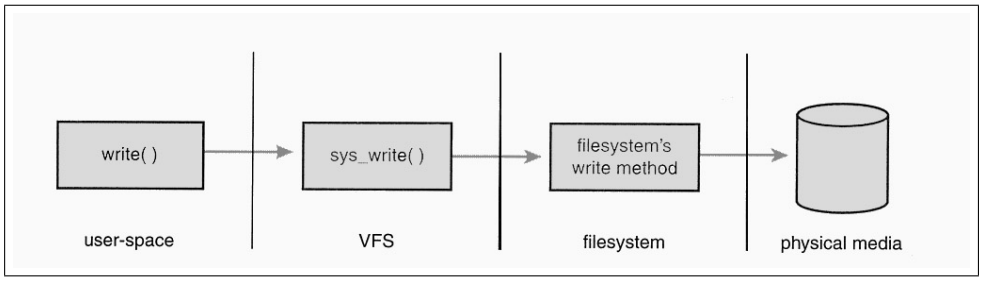

Figure 3.7: Issuing a write() call [51]

3.4.1 Common File Model

Even though the kernel is programmed in C, which is not an object-oriented language, the VFS is in fact object-oriented. The VFS objects are implemented as plain C data structures, where some of the fields contain pointers to the file system-implemented functions corresponding to the object's methods [32].

VFS uses a *common file model* capable of representing all supported file sys-

tems [32]. The common file model is based on the abstractions found in the traditional Unix file system: files, directory entries, inodes, and mount points. Thus, the four primary object types of the VFS are [51]:

- The *superblock* object, representing a specific mounted file system.
- The *inode* object, representing a specific file.
- The *dentry* object, representing a directory entry, a single component of a path.
- The *file* object, representing an open file as associated with a process.

Each of the four primary objects types described above contains an *operations* object, which describes the methods that the kernel can invoke in the primary objects [51]. Details of the four primary objects, and the operations objects associated with them, can be found in Appendix B.

Figure 3.8 [32] helps in illustrating how the concepts and objects of VFS fit together. Three different processes have opened the same file, process 1 and process 2 use the same hard link[2], while process 3 uses a different hard link. Each process has its own file object, but only two dentry objects are needed, one for each hard link. The dentry objects refer to the same inode object, which identifies the superblock object. The superblock object and the inode object together identify the common disk file.

3.4.2 Proc - a special file system

Linux supports a wide range of file systems, and while most of them are conventional file systems, some are treated in a special way.

The */proc* file system is a virtual file system that acts as an interface to internal data structures in the kernel. Each file in the */proc* file system is tied to a kernel function that simulates reading or writing from a real file [51].

The original purpose of the */proc* file system was to provide information about the processes running in the system, but */proc* now also provides information about a wide variety of other system data. This has made the kernel developers view the */proc* file system as a bit of an uncontrolled mess [36], and caused the development

[2] A filename included in a directory is called a file *hard link*. Because more than one hard link can be associated with the same file, a single file may have several filenames.

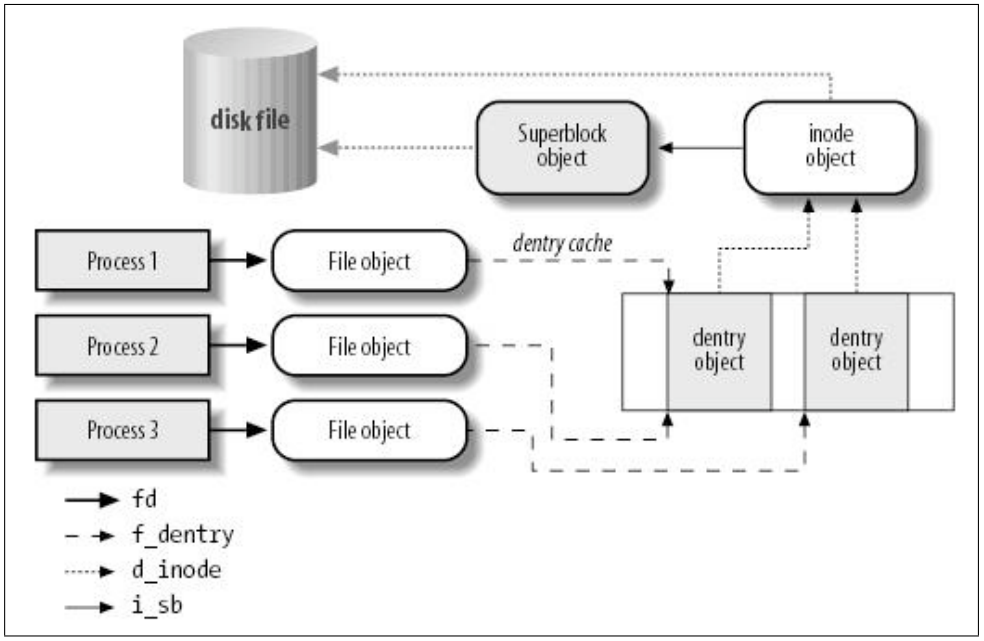

Figure 3.8: Interaction between processes and VFS objects [32]

of another special file system, *sysfs*, which is tasked with providing information about the system's devices, thus reducing some of the clutter in */proc*.

Because */proc* provides information about the processes running on the system, it is used by a variety of utilities. For example, *ps*, *top* and *uptime* all gather their information from */proc*. By modifying */proc*, the visibility of individual processes can be manipulated, a technique used by some rootkits (see section 4.4.2).

/proc contains one subdirectory for each process running on the system, named after the process ID (PID). Table 3.1 lists some of the entries common to all processes.

The data structures of the */proc* file system in special, and the whole VFS in general, all represent important targets for malicious code. As described in chapter 4, some of the existing kernel level malware used today are able to subvert the kernel by modifying these structures.

File	Content
cmdline	Command line arguments
cpu	Current and last cpu in wich it was executed
cwd	Link to the current working directory
environ	Values of environment variables
exe	Link to the executable of this process
fd	Directory, which contains all file descriptors
maps	Memory maps to executables and library files
mem	Memory held by this process
root	Link to the root directory of this process
stat	Process status
statm	Process memory status information
status	Process status in human readable form
wchan	If CONFIG_KALLSYMS is set, a pre-decoded wchan
smaps	Extension based on maps, presenting the rss size for each mapped file

Table 3.1: Process specific entries in */proc*

Chapter 4

Malicious kernel code

4.1 Introduction

This chapter will give the reader an introduction to some of the concepts of malicious kernel code. Most often this type of code comes in the form of rootkits. As malicious kernel code and rootkits pose a great threat to integrity of the kernel, they deserve a chapter of their own.

Many definitions of rootkits may be found: Levine et al. [49] argue that rootkits may be characterized as trojan horses. A trojan computer program is usually defined as something like: "a program with an overt (documented or known) effect and a covert (undocumented or unexpected) effect" [31]. By calling a rootkit a trojan, one also implies that the rootkit has an overt effect. This may not always be correct as the rootkit by nature tries to hide its presence from the user. On the other hand the rootkit provides functionality to the persons that place a rootkit on a system. From their point of view it may be correct to characterize it as a trojan.

Another definition is given by the SANS (SysAdmin, Audit, Network, Security) Institute [7]:

> A collection of tools (programs) that a hacker[1] uses to mask intrusion obtain administrator-level access to a computer or computer network

Using this definition, the rootkit is a toolkit that is installed on a computer system after an attacker has gained administrative access to it. The means by which

[1]The word "hacker" is much debated. This thesis uses the word hacker as meaning one that is utilizing malicious code. (A "cracker" might be a better suited, but perhaps less well known term).

such access is gained is not of interest in this context. Once installed, rootkits modify the host's software to provide an attacker with the ability to hide the existence of chosen processes, files or network connections from other users. They may also provide convenient back doors through which an attacker may regain privileged access to the host, or even keystroke logging facilities for spying on other users.

The goal of this chapter is to give the reader insight in how this type of malicious code works, and why it is difficult to detect. Different detection techniques will be described, followed by a general discussion of their use. At the end we will try to sum up what seams to be the most promising techniques to counter the threats posed by malicious code.

4.2 Background

In short, the reason rootkits were first developed, was because breaking into a computer system usually is a lot of hard work. Attackers therefore wanted to make sure they didn't have to do all this work every time they wanted to break into a specific system. The rootkit mainly functioned as a back door into the system. The rest of the rootkit was mostly meant to provide means to keep this backdoor hidden on the computer. Since the arrival of the first rootkits, they have evolved a lot both in terms of functionality and techniques [40]. This evolution has resulted in the development of continuously more sophisticated countermeasures, leading to what one might call an arms race between the "good" and the "bad" guys. In this race, the "bad" guys, meaning the rootkit creators, have traditionally held the advantage [57].

4.3 The threat

According to Greg Hoglund, one of the leading experts on rootkits, rootkits are not only used for malicious activities. In his book "Rootkits - Subverting the Windows kernel" [40], he gives multiple examples of how rootkits can be used with good intentions by e.g the government in areas like military warfare, computer security and crime investigations.

Also, there have been examples of rootkit technology being used by commercial actors as a technique to implement digital rights management systems. The best known example is Mark Russinovich's disclosure of Sony's use of the Aries rootkit shipped along with many of their music CDs [64]. Russinovich also pointed out that two of the most popular CD emulation utilities, *Alcohol* and *Daemon Tools*, both make use of rootkit technology [65]. According to a senior official in Mi-

crosoft Corp.'s security unit [55], by December 2005 more than 20 percent of all malware removed from Windows XP SP2 (Service Pack 2) systems were stealth rootkits.

So, again according to Hoglund, rootkits are not inherently bad, they are just a technology. Since the cornerstone of this technology is keeping things hidden from a administrator of a computer system, it is important to have knowledge of the possible severe consequences they may have on the infected system.

4.4 Types of rootkits

Rootkits are usually partitioned into two main categories: *user mode*, and *kernel mode* rootkits [71]. The former is also known as first generation rootkits [40]. The main difference is that the former does not modify the attacked kernel's code and the latter does.

4.4.1 User mode rootkits

These rootkits usually replace critical system utilities such as *ps*, *ls* and *netstat* with versions that work in favor of the attacker by filtering information returned from the kernel. In this way an attacker may succeed in hiding files, processes and network connections of his liking.

Since the kernel in this case remains unmodified, these rootkits are easily disclosed by an suspicious administrator by comparing the output from the modified versions of the *ps*, *ls* etc. with unmodified versions of these programs. The correct information may also be obtained by asking the kernel directly through the *proc* file system, as described in section 3.4. It is also quite common for an administrator to monitor important system files with an integrity checker like Tripwire [20] or AIDE [2].

4.4.2 Kernel mode rootkits

This type of rootkits penetrates the kernel of the running operating system. Once inside the kernel they are pretty much able to do anything they like. The next sections describe the most common techniques rootkits of this type use to make the kernel return incomplete or wrong information when queried by user mode programs. Figure 4.1 [57] sums up most of the techniques used by the most common rootkits today. There may be more known and unknown techniques than the ones listed below, but these are the most used and widely discussed techniques that the authors have found.

rootkit name:	loads via:	overwrites syscall jump	adds new syscall jump	modifies kernel text	adds hook to /proc	adds inet protocol
Complete rootkits:						
adore 0.42	LKM	x				
knark 2.4.3	LKM	x			x	x
rial	LKM	x				
rkit 1.01	LKM	x				
SucKIT 1.3b	kmem	x	x			
synapsys 0.4	LKM	x				
Demonstrates module or process hiding only:						
modhide1	LKM	x				
phantasmagoria	LKM			x		
phide	LKM	x				
Demonstrates privilege escalation backdoor only:						
kbd 3.0	LKM	x				
taskigt	LKM				x	
Demonstrates key logging only:						
Linspy v2beta2	LKM	x				

Figure 4.1: Rootkits and their techniques
[57]

Entering the kernel

To be able to modify the kernel, the rootkit first have to get access to the kernel symbols and structures. This is usually achieved in one of two ways: Through the loading of kernel modules or by writing directly to the memory of the running kernel via the */dev/kmem*[2] interface [71].

The use of loadable kernel modules (LKMs) is a very useful feature of the Linux kernel, but it also serves as the entry point of most of today's rootkits [71]. By constructing the rootkit as an LKM, the rootkit is easily loaded into the kernel and thereby automatically given full access to all of the kernel. From here it is only up to the rootkit creator what kernel code he wants to modify or add.

The */dev/kmem* device file is a special character device in the linux kernel that enables user mode processes with administrator privileges to access memory as it was a regular file. SucKIT [18] is a rootkit which inserts itself into the kernel by patching */dev/kmem*, effectively writing itself into kernel memory.

Surviving a reboot

When the rootkit inserts itself into kernel memory, it is lost if the system reboots. Some rootkits try to avoid this by not acting as standalone LKMs, but insted inject their code into a suitable kernel module already installed on the system. By piggybacking on another module, the rootkit is automatically reloaded into the kernel

[2]The /dev/kmem provides read/write access to the virtual memory space. /dev/mem provides the same access to physical memory.

during the boot sequence. To accomplish this the rootkit has to modify a binary LKM file on the disk, and the attacker therefore would be an easy prey for an attentive integrity checker.

Common rootkit techniques

Below follows a description of the most common techniques used by rootkits.

Overwriting syscall jumps This is the most common technique utilized by rootkits, as can be seen in figure 4.1. Overwriting a syscall jump simply means to change one of the addresses in the system call table to point to a function provided by the rootkit (see section 3.3.6). With this technique a rootkit can intercept a system call, run its own code and then, if it wants, call the original function. This is mostly used to filter information when a user process, such as *ls*, requests information from the kernel [71]. By filtering the information returned by e.g the system call *sys_getdents*, the rootkit is able to hide certain files or folders. Being the most commonly used method, this is also the one thing all rootkit detectors try to look for (see section 4.5).

Adding new syscall jumps Since most rootkits is detected when modifying the system call table, some rootkits use a technique where they instead modify the entry in the Interrupt Descriptor Table (IDT) used in system call invocations. As described in section 3.3.5, a system call can be invoked by issuing a software interrupt using the `int $0x80` assembly instruction. The software interrupt handler is defined in the IDT, and by replacing this handler a new system call table can be created. In this way the original syscall table is left unchanged, and anyone checking for changes in the original table will find nothing unusual.

Modifying kernel text In this context, the *kernel text* refers to the code segment of the kernel resident in memory (see section 3.3.3). By changing this code, a rootkit changes the application logic of the kernel code. This can lead to an almost infinite range of new behaviour in the kernel. The *phantasmagoria* rootkit uses this technique to change the code of the kernel's scheduling mechanism, allowing processes to be scheduled for processor time, but at the same time removing them from the kernel's list of currently running processes.

Hooking Hooking means to alter the call chain invoked by a kernel procedure to include one of its own functions in the chain [23]. This is a general technique that can be applied in many places in the kernel. The *taskigt* uses this to

grant administrative privileges to any process that reads from a particular *proc* entry [57].

Manipulating the Virtual File System This is a technique utilized by the newest version of *adore*: *adore-ng* [1]. *Adore-ng* modifies the *proc* file system so that the rootkit's own functions is called whenever someone reads from *proc* entries. By doing so, *Adore-ng* is able to hide processes from view.

Adding inet protocol handler [3] This means that the rootkits registers its own handler to process special packets arriving at the network interface. The rootkit manipulates the TCP/IP stack by changing the original network protcol handler with a version of its own. When a packet comes in, the rootkit analyzes the packet, if it is "evil", it is processed by the rootkit, if not, the original handler is called [71]. The *knark* [10] rootkit uses this functionality to spawn privileged processes running programs of its own choice, when certain kinds of "evil" packets are received [57].

4.5 Detecting kernel mode rootkits

As described in the section about rootkit techniques(4.4.2) above, these kinds of attacks can be very hard to detect, and sometimes the attacker may be left undetected for months and years before he is detected [71].

4.5.1 Preventing entry

The best prevention would be if the rootkits were never allowed to enter a system. The problem is knowing which "doors" to guard. The two main entry points are */dev/kmem* and LKMs, as was shown in section4.4.2.

Kruegel et al. describes how malicious modules can be detected before being loaded into the kernel, by doing static binary analysis of the module binaries [47]. This approach is based on the observation that the runtime behaviour of regular kernel modules differs significantly from the behaviour of kernel-level rootkits, and that the behaviour can be determined trough binary analysis.

To counter the use of the */dev/kmem* entry point, the Red Hat Linux distribution removed this special character device from its kernel some time ago, but it is still present in most other distributions. Even if */dev/kmem* is removed, it is still possible to write to */dev/mem*. This device file is not easily removed, as several

[3] inet(Internet protocol family) is a collection of protocols layered atop the Internet Protocol (IP) transport layer, and utilizing the Internet address format.

applications rely on this device (an example is the XFree86 window system, which uses */dev/mem* to access the video card). And even if both */dev/kmem* and */dev/mem* are removed, there are still techniques that a privileged user can use to access the memory. More details regarding those techniques can be found in [71].

4.5.2 Searching for anomalies

Rootkits often leave traces to their whereabouts. Detecting them is often just a question of knowing where to look. One might look for anomalies like unusual network traffic, open ports, network cards in promiscuous mode, abnormal disk usage, log file entries, mismatches between hard link counts and the output from *ls* [71].

Saint Jude is a tool that is much used for anomaly detection [33]. It is self learning in the sense that it first observes the normal behaviour of the system, then creates a set of rule based policies based on the observations. The system is then monitored for behaviour that violates the policies created.

Most of the rootkits have very specific behaviours that may reveal their existence on a system. One well known example is how *knark* makes use of the signal mechanism in Linux. A *signal* is a short message that may be sent to a process, used to make the process aware that a specific event has occured [32]. Signal number 31 is normally unused, but if a process receives this signal when *knark* is installed, the process gets hidden. The use of signal 31 is thus a good indication of the presence of *knark*.

Examining call execution

Patchfinder 2 is a tool for the Windows family of operating systems, which examines the execution of call sequences in the kernel functions [67]. The number of calls made in each function is recorded as *Patchfinder* starts up. Then the number of calls made is periodically recorded as the system is running, and compared to the ones in the baseline to reveal any discrepancies.

4.5.3 Hidden files and processes

As mentioned at the start of this chapter, most rootkits include the capability to hide files, directories and processes. Detecting these hidden resources can be done by directly obtaining information from kernel structures, without relying on the possible compromised system calls of the operating system.

This technique has been proven effective by another tool for the Windows operating systems, *RootkitRevealer* [66]. *RootkitRevealer* compares the information

provided through the Windows API with the information obtained from the raw contents of the file system.

Another technique used for detecting hidden processes, is using a kernel module for hooking into the kernels scheduling routine. Because the hidden processes must be given processor time by the operating system's scheduler in order to be useful to the attacker, the scheduler have to get some knowledge about them at some time. By hooking into the scheduler, it can be detected if the scheduler has been modified to also schedule processes that were previously removed from the *task_array*[4] list [34].

4.5.4 Hidden kernel modules

The modules running in the kernel are kept in a linked list of module objects, aptly named *modules*. To locate kernel modules that a rootkit has removed from the *modules* list, one may search the memory area for objects that seem to have the structural layout of a module [71]. In Phrack Magazine #61 [53], a kernel module named *MODULE_HUNTER* was presented, using such a technique to search the kernel for hidden modules.

4.5.5 Integrity checking the kernel memory

Information about the memory of the running kernel can normally be extracted from two places: The */dev/kmem* special device file, and using a module loaded into the kernel [34]. Neither of these options is ideal, as they both operate within an environment that may already be subverted, and are both already dependent of the integrity of the kernel they are to check. Nevertheless, several tools performing integrity checks of the kernel from within the (possibly compromised) system exist [15] [33] [17].

The approach usually taken by integrity checking tools, is to obtain a copy of important kernel structures at startup, comparing the subsequent memory scans to it. Much of the memory of a running system is highly dynamic. Identifying which parts of memory to check is therefore one of the biggest challenges in this type of rootkit detection. Below is a listing of some important kernel structures that are not expected to change during system execution:

Kernel text This is the area of memory containing the compiled kernel code.

[4] an array in the Linux kernel that contains pointers to all the process descriptors (see section 4.4.2

LKM text The code of LKMs resides in dynamically allocated memory areas, that are allocated when the LKMs are loaded into the kernel and freed when the LKMs are unloaded from the kernel.

System call table This structure is targeted by most of the rootkits we know of today, and should not be modified in a healthy kernel.

Interrupt descriptor table This is the place to make changes if a rootkit wants to trick the kernel into using a different system call table. This table also handles the other kernel interrupts, and should be kept static.

Virtual File System While the VFS is highly dynamic, allowing for run-time registration of the function handlers associated with the particular file systems, those handlers are usually not expected to change after first being registered. As an example, the function handlers used for the operations of the */proc* file system, should remain static through the exectution of the system.

4.6 Discussion of the detection methods

The ways of detecting rootkits described above all have their pros and cons. Below, some positive and negative aspects of the various methods will be discussed.

The static binary analysis used by Kruegel et al. [47], is not only a detection method, it is also a prevention mechanism as the malicious code is detected prior entering the kernel. In their testing, almost 1000 kernel modules were analysed; no false positives were detected, and 8 commonly known rootkits were correctly identified. While the numbers are impressive, it is an open question if it would be possible for a rootkit developer to fool the detection algorithm given the details of how it works. Also, the algorithm used for binary analysis is based on the behaviour of existing rootkits, its usefulnes in providing protection against new techniques can thus be questioned.

The main problem with the anomaly detection techniques described in section 4.5.2, seems to be that they all rely on the attacker having made some sort of a mistake when designing his rootkit, causing the system to behave in an anomalous way. While it is certainly true for some rootkits, it is unlikely to be true for all.

Checking for known anomalies caused by a particular rootkit, is in fact a form of signature detection. As such, it has the same properties of being effective against known threats but as good as useless against new threats.

Rutkowska's findings when examining the call chains of kernel functions, showed that it was indeed possible to detect abnormal chains of function calls, which indicated that the kernel had been subverted. This method does however suffer from the natural variations in the execution of functions, caused by conditional statements within the code, making it crucial to define what should be considered acceptable variations within the monitored system [40].

Integrity checking of vital kernel structures, seems like a sound approach, capable of detecting both known and unknown rootkits. The approach is not without challenges, as the structure vital for the kernel integrity needs to be identified, and a reliable way of inspecting their state must be provided.

Most implementations are limited not only by the techniques they use, but of the placement of the detection mechanism. By placing the detection mechanism within the system to be monitored, the integrity of the mechanism itself and of the data on which it operates, can be questioned. Some of the research efforts described in section 2.4 address this problem, using various methods for ensuring both the safety of the detection mechanism and the correctness of the data used by it. The Copilot project is one example [57], using a PCI card based monitor. Another example from section 2.4 is the Livewire IDS [37], which uses virtual machines for securing the detection mechanism, while maintaining access to the entire hardware state of the monitored host. The suitability of virtualization as a platform for doing integrity analysis, is further explored throughout part II of this thesis.

Chapter 5

Virtualization

5.1 Introduction

Today, the term virtualization is used to cover a wide range of ideas, concepts and technologies.

The virtual memory abstraction found in all recent Unix and Linux systems, is a good example of how modern operating systems are already using virtualization techniques (see section 3.3.3). Also, in the multiprogramming model used by most modern operating system, each process can be thought of as having its own virtual processor while they in reality share a single physical processor.

Virtualization encompasses such a broad range of ideas, that it is hard to provide a strict, formal definition of it. Nanda and Chitueh [74] define virtualization as: "(...) a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others." Singh provides a similar definition [73]: "Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.", or as he later puts it: "virtualization abstracts out things."

5.2 Virtual machines

A key concept of virtualization in all its variations, is the concept of the virtual machine. The virtual machine is a software abstraction, for which we write software to run upon.

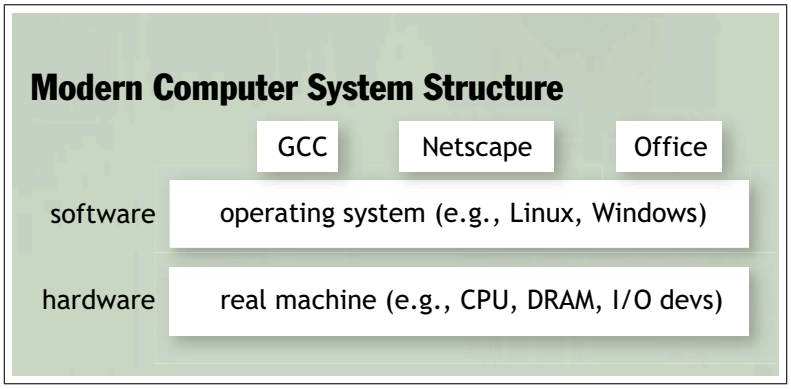

Figure 5.1: Computer system structure [63]

Figure 5.1 shows how modern computing systems are composed of layers, consisting of the hardware, an operating system and application programs running on top of the operating system.

Virtualization software abstracts virtual machines by interposing a layer at various places in the system [63]. Using this notion of *virtualization layers*, the set of virtualization technologies can be partitioned into three types:

1. Hardware-level virtualization

   This is the traditional form of virtualization, where the virtualization layer is located right on top of the hardware, resembling a real machine (see figure 5.2). By replicating the hardware, all software written for it will run in the virtual machine, effectively allowing multiple operating systems to be run concurrently. Examples include VMware Server/Workstation, Xen and Parallels.

2. Operating system-level virtualization

   Here the virtualization layer is situated between the operating system and the application programs. The virtual machine environments are all running on the host operating system, sharing the same kernel. This is sometimes referred to as lightweight virtualization or container-based virtualization. Some examples are FreeBSD Jails, Solaris Containers and OpenVZ.

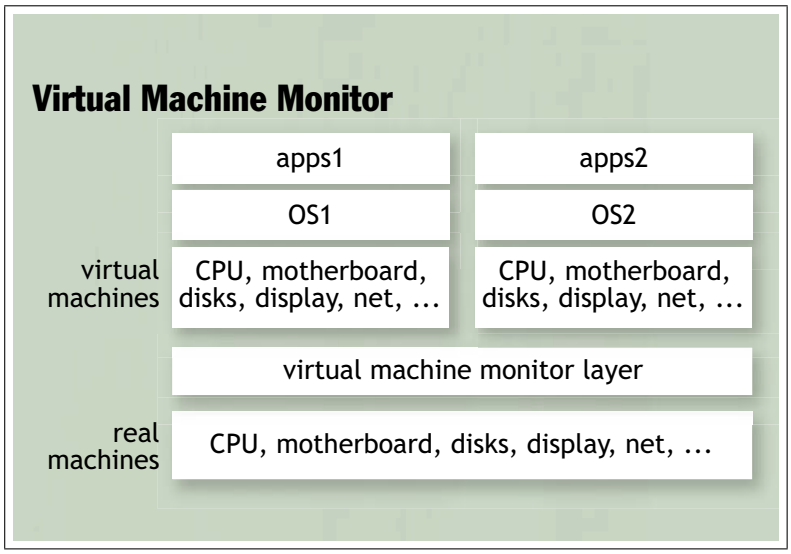

Figure 5.2: Hardware-level virtualization [63]

3. High-level language virtual machines

   This type places the virtualization layer on top of the operating system, as an application program. The virtual machine implements an abstract machine definition, allowing application written in the high-level language and compiled for the virtual machine to run on it. The Java Virtual Machine is an example of this kind of virtual machine.

5.3 Hardware-level virtualization

In this thesis, virtualization will be employed as a tool for checking the integrity of the operating system's kernel, making the hardware-level the appropriate level of virtualization.

5.3.1 Virtual machine monitors

In their classic 1974 paper [59], Popek and Goldberg define a virtual machine (VM) as: "an efficient, isolated duplicate of the real machine". Multiple VMs can be run concurrently on a single hardware platform, each instance running an operating system of its own.

The Virtual Machine Monitor (VMM) is the system software responsible for creating and controlling the virtual machines, and the environment in which they run.

Popek and Goldberg identified three essential characteristics of a VMM:

1. The VMM provides an environment for programs which is essentially identical to the original machine.
2. Programs that run in this environment show at worst only minor decreases in speed.
3. The VMM is in complete control of the system resources.

The first of these characteristics ensures that a program executing on a virtual machine runs the same as it would if it was running directly on the original machine. There are, however, some exceptions to this rule. Differences caused by the availability of systems resources, such as the amount of available memory, may lead to the program performing differently. The intervening nature of the VMM may also cause differences related to timing dependencies. The second characteristic is an efficiency requirement, and is what separates VMMs from hardware emulators and simulators. Most of the virtual processor's instructions must therefore be executed directly by the real processor without the intervention of the VMM. The third characteristic is that of resource control. The VMM should ensure that none of the VMs can access any resource not explicitly allocated to them, and that the VMM can regain control of previously allocated resources.

5.3.2 Virtual machine environments

The virtual machine systems of today typically use one of two different approaches to build the virtual machine environment. In *type I* environments, the VMM runs directly on the machine hardware (see figure 5.3).

The type I VMM, also known as the standalone VMM, can be thought of as an operating system kernel with additional mechanisms to support virtual machines. It requires drivers for hardware peripherals, and performs scheduling and resource allocation for the virtual machines. Typically, a type I VMM is very small, making it easier to ensure the robustness and security properties of the VMM.

In *type II* environments, however, the VMM runs as a a normal process on a host operating system, as shown in figure 5.4. It is thus able to take advantage of the host operating system for memory management, processor scheduling, resource allocation, and hardware drivers [61]. However, the added comfort provided by the type II environment, comes with a price: Security vulnerabilites in the host OS, will also render the VMM and the virtual machines vulnerable.

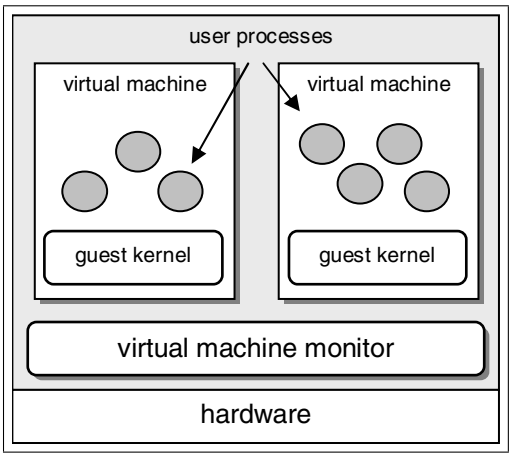

Figure 5.3: Type I virtual machine environment [48]

5.3.3 Virtualization techniques

In this section, the classical techniques used in implementing VMMs will be described, and some obstacles posed by the x86 architecture will discussed. Finally, some extensions to the x86 architecture that allow for hardware-assisted virtualization, will be described.

Architectural requirements

Robin and Irvine [61] cite Goldberg [38] as having identified the key architectural features of hardware pertitent to virtual machines:

- two processor modes of operation,
- a method for non-privileged programs to call privileged system routines,
- a memory relocation or protection mechanism such as segmentation or paging, and
- asynchronous interrupts to allow the I/O system to communicate with the CPU.

As stated above, the processor needs at least two modes of operation, a privileged mode and a unprivileged mode. In the privileged mode, the complete instruction set is available for the processor, while in unprivileged mode, only a subset is. When instructions that read or write privileged state execute when the processor is in unprivileged mode, they are made to trap. When a trap occurs, the instruction is

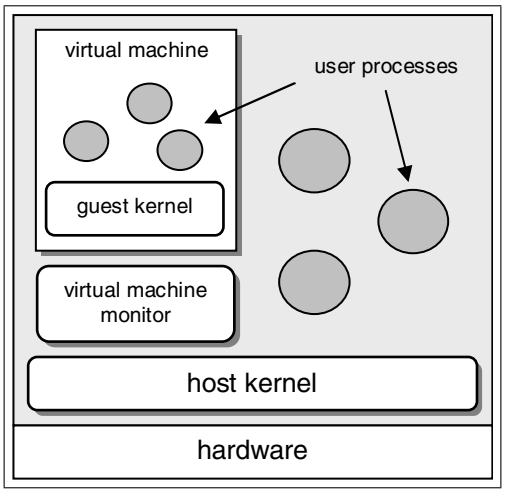

Figure 5.4: Type II virtual machine environment [48]

passed on to the VMM, which emulates the trapping instruction against the virtual machine state. This technique is known as *trap-and-emulate* [28].

Some sort of memory protection mechanism is necessary to deny guest accesses to in-memory privileged state. VMMs typically maintain *shadow page tables* for use by the guest, using hardware page protection mechanisms to trap and emulate memory accesses [28].

The x86 architecture

As mentioned above, some processor instruction should not be executed directly on the processor because they expose privileged state. These instructions are called sensitive instructions. For an architecture to be virtualizable, sensitive instructions must trap when executed in unprivileged mode, or as Popek and Goldberg [59] puts it: "(...) a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions". The x86 architecture does, however, contain several sensitive, unprivileged instructions. After examining the instruction set of the fifth generation x86 processor (the Pentium), Robin and Irvine [61] identified a total of seventeen such instructions.

As a result, the lack of full virtualization support in the x86 architecture has necessitated the development of alternative approaches.

Binary translation. Executing instructions on an interpreter as opposed to a physical processor, bypasses the problems of x86 virtualization. An interpreter

could prevent leakage of privileged state, and correcty implement the sensitive non-privileged instructions [28]. However, interpretation does not fullfill the performance requirement of Popek and Goldberg; an interpreter using a fetch-decode-execute cycle could use hundreds of actual physical instructions per guest instruction. *Binary translation*, however, operates on larger pieces of code and caches the results for future use. By using a suitable binary translator, the performance requirement may be met.

Adams and Agesen [28] describe the binary translator used in VMware as having the following properties:

- *Binary.* Input is binary x86 code, not source code.
- *Dynamic.* Translation happens at runtime.
- *On demand.* Code is translated only when it is about to execute.
- *System level.* The translator makes no assumptions about the guest code.
- *Subsetting.* While the translator's input is the full x86 instruction set, the output is only a safe subset.
- *Adaptive.* The translated code is adjusted in response to guest behaviour changes, improving overall efficiency.

Paravirtualization. A different approach in virtualizing the x86 architecture, is to relax the requirements of full virtualization, and use a virtual machine abstraction that is similar but not identical to the underlying hardware platform [30]. This approach, which has been dubbed paravirtualization, is the approach taken by Xen (see chapter 6).

Paravirtualization promises higher performance and a simpler VMM, but because of the virtual machine not being identical to the underlying hardware, paravirtualization does require modification of the guest operating system. Because of this, the only operating systems that are likely to be supported, are those with an open source license.

x86 architecture extensions. Both Intel and AMD recently introduced architectural extensions to their x86 processors, making classical trap-and-emulate virtualization possible on the x86 architecture.

Adams and Agesen [28] have compared the performance of a VMM implementation using these extensions with the performance offered by their BT-based software VMM. For some workloads the hardware VMM outperforms the software VMM, but, perhaps surprisingly, not for all. This may suggest that the recent hardware support for virtualization will be complementing existing software techniques, rather than replacing them completely.

5.4 Virtualization solutions

There are many virtual machine solutions available today. Here, only a brief overview of some of the most prevalent ones will be given.

5.4.1 VMware

VMware has several virtualization products, VMware workstation and VMware ESX server being the most interesting ones.

VMware workstation is a type II VM, running within a host operating system, while VMware ESX server is a standalone, type I VM, running directly on top of the hardware.

Both provide full virtualization through binary translation, and are capable of running most x86 operating systems as guests. The VMware products are distributed as binaries, with no source code available.

5.4.2 Xen

Xen is an open source virtual machine monitor, originally created at the University of Cambridge Computer Laboratory.

As described above, Xen uses paravirtualization to achieve high performance. Xen has since version 3 also supported classical virtualization using the recent x86 architectural extensions, thus enabling unmodified operating systems to run on it. Chapter 6 will provide the reader with a more detailed description of the Xen hypervisor.

5.4.3 Parallels

While Parallels are perhaps best known for their Desktop for Mac product, allowing Microsoft Windows to be run on Mac OS X, they also offer Parallels Workstation for use on Microsoft Windows or Linux.

While Parallels use a lightweight hypervisor concept similar to Xen, they do not use paravirtualization, instead relying on the x86 architecture extensions from Intel.

5.4.4 KVM

KVM is a type II virtualization environment for Linux, using the x86 extensions from AMD and Intel to achieve full virtualization capabilities.

As shown in figure 5.5, KVM is implemented as a Linux kernel module, using Linux as the hypervisor. Even though KVM is a rather recent effort, it has already been included in the mainline Linux kernel tree, giving it considerable momentum.

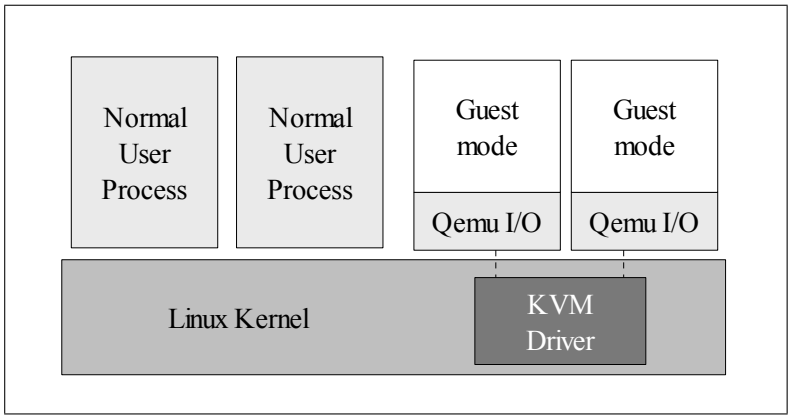

Figure 5.5: A kvm based architecture [43]

5.5 Virtualization and security

Some properties of virtualization are especially relevant to the field of information security.

5.5.1 Isolation

The various variants of virtualization offer differing degrees of isolation. Virtual machines running in operating system-level virtualization solutions, share the same kernel as their host systems, and a compromise of the kernel will affect the host system and all the virtual machines running there.

Hardware-level virtualization provides strong isolation, with each virtual machine running in its own hardware protection domain. Code running in a virtual

machine cannot access code running in the hypervisor or in some other virtual machine.

5.5.2 Software reliability

The correctness of the VMM is fundamental to the issue of reliability. As Goldberg [39] points out, it is a reasonable assumption that the VMM is correct, because the VMM is likely to be a very small program with limited functionality. While this certainly holds true for a Type I VMM, the reliability of a Type II VMM is highly dependent on the underlying host operating system. As Robin and Irvine [61] observe, flaws in host OS design and implementation will render the virtual machine monitor and all its virtual machines vulnerable.

Chapter 6

The Xen virtualization technology

We have chosen Xen as the virtualization solution for this thesis. This chapter will describe the concepts and architecture of Xen, focusing on the issues most relevant to this thesis. Xen, starting out as a research project at the University of Cambridge, is today being actively developed by a large community of open source developers.

Section 6.1 gives an overview of the Xen architecture, while section 6.2 takes an in-depth look at the Xen hypervisor.

6.1 Architectural overview

Xen has become widely known for its use of the paravirtualization technique, which obtains high performance by modifying the guest operating systems. Since version 3.0, Xen also provides support for unmodified guest operating systems, using the x86 architecture extensions from Intel and AMD (see section 5.3.3, page 47). Figure 6.1 gives an architectural overview of Xen 3.0.

In the figure, four virtual machines run on top of a virtual machine monitor, termed the Xen Hypervisor, or just the *hypervisor* for short. Xen's hypervisor is quite small, providing only basic control operations [30]. The virtual machines are identified as VM0-4, but in Xen-terminology the term *domain* is often used to refer to a virtual machine. The following gives a short description of all of the components shown in the figure:

VM0 is the initial virtual machine, and is created automatically at boot time. It is usually referred to as *domain 0* [30], or just *dom0*. Domain 0 is a required

part of any Xen-based server and runs the application software that manages Xen. The operating system used in VM0 is a modified version of Linux.

VM1-3 are unprivileged domains, usually referred to as *domUs*. In this figure VM1 and VM2 run paravirtualized version of Linux, while VM3 runs an unmodified guest operating system using the x86 architecture extensions of either Intel (VT) or AMD (AMDV). The VT and AMDV extensions are so similar that Xen has developed a common interface layer, the Hardware Virtual Machine (HVM) layer, and domains running unmodified operating systems are often referred to as HVMs.

Split device driver The figure also depicts Xen's *split device driver* architecture. Xen uses two co-operating drivers to provide the illusion of a virtual device, the *frontend* driver and the *backend* driver. The frontend drivers runs in the unpriviliged domains, while the backend drivers run in domain 0, which is usually the only domain with access to the real device hardware.

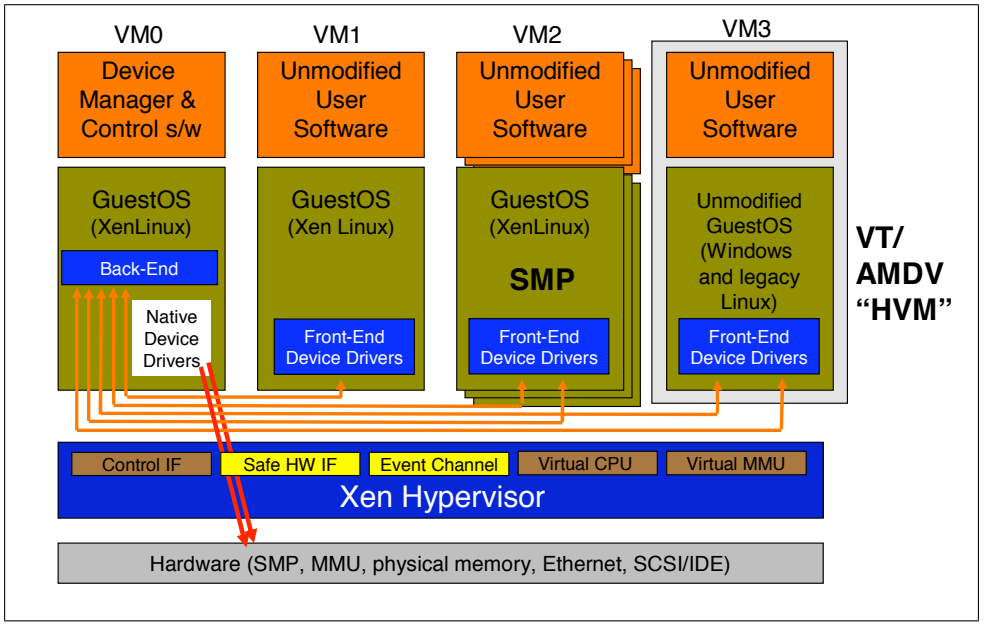

Figure 6.1: Xen architecture [60]

6.2 Xen - the hypervisor

6.2.1 CPU

The protection model of the x86 architecture consists of four privilege levels [44], where privilege level 0 is the highest and privilege level 3 the lowest. The privilege levels can be interpreted as rings of protection, as depicted in figure 6.2.

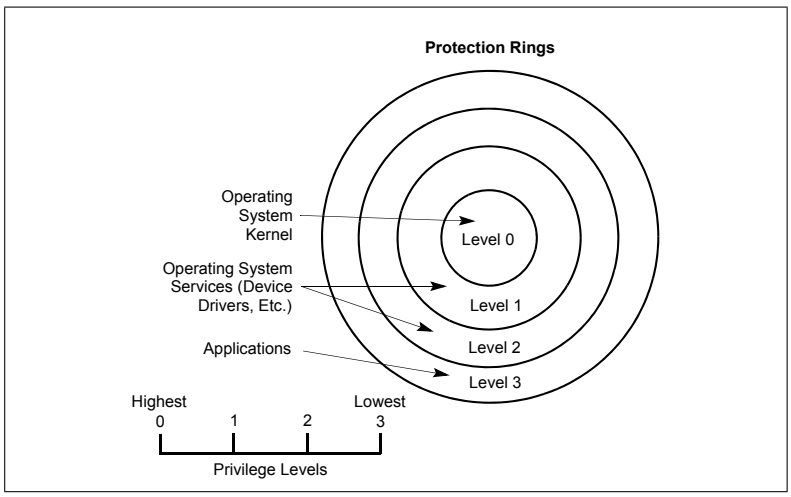

Figure 6.2: x86 protection rings [44]

Even though four rings are provided, most modern operating systems use only two. The kernel code typically runs in ring 0, because this is the only ring that can execute privileged instructions, while application code runs in ring 3.

In Xen, the hypervisor runs in ring 0, while the guest operating systems are demoted to run in ring 1. The applications still run in ring 3 (see figure 6.3). This means that all privileged instructions, such as installing a new page table, or yielding the processor when idle, must be handled by Xen. The guest operating systems use *hypercalls* to invoke operations in Xen, analogous to the use of system calls in conventional operating systems. As described in section 3.3.6, a system call is a way to move from user space (ring 3) to kernel space (ring 0). Likewise, hypercalls are used in order to move from ring 1, where the guest OS runs, to ring 0, where Xen runs.

As an example of the use of hypercalls, a guest OS might make a hypercall to request a set of page-table updates, which Xen then validates and executes before returning control to the guest OS [30].

Paravirtualization makes it possible to implement various optimizations by

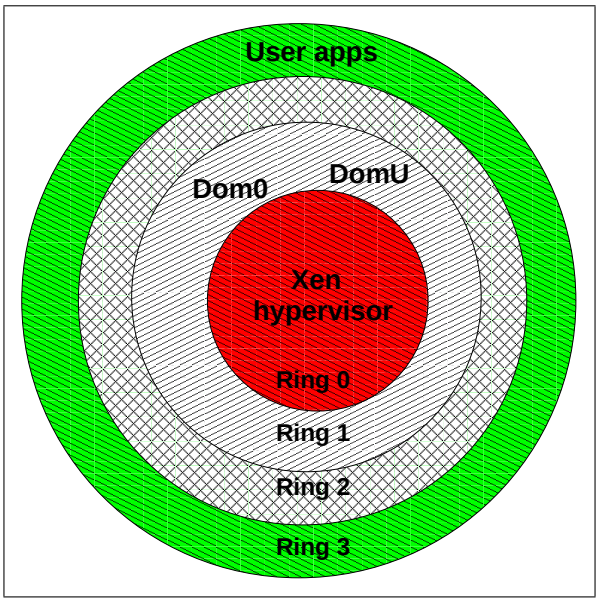

Figure 6.3: Xen and the protection rings

modifying the guest OS. The system call mechanism provided by an operating system is usually implemented via the use of a software exception, which would necessitate a trap into Xen, since the OS is running in ring 1. By allowing the guest OS to register an exception handler that is accessed directly by the processor without indirecting via ring 0, the performance of system calls is greatly improved [30].

Not all privileged instructions are replaced by hypercalls, some instead use the classical trap-and-emulate technique described in section 5.3.3.

6.2.2 Event channels

Communication from Xen to a domain is provided through an asynchronous event mechanism, called *event channels*. Events are the Xen equivalents of hardware interrupts, and the guest OS can map these events onto its normal interrupt dispatch mechanisms. Just as hardware interrupts can be disabled on the processor, events too may be disabled by setting a software flag [25] [30].

6.2.3 Memory

The complexity of memory management in modern computer systems, makes virtualizing memory a challenging task. Traditionally, this has been solved through the use of *shadow page tables*, where the hypervisor provides the guest OS with an

independent copy of page tables, not visible to the hardware memory-management unit (MMU).

Although Xen does support shadow page tables, it uses a different approach in its default configuration. By registering the guest OS page tables directly with the MMU, and giving the guest OS read-only access to them, Xen avoids the complexity and overhead from using shadow page tables. When the guest OS needs to make an update to a page table, it issues a request to Xen via a hypercall. Xen then validates the request, and applies the update if it is deemed safe.

Physical memory

Xen needs only a small portion of physical memory for its own use, the rest of the physical memory is available for allocation to domains. Physical memory is always allocated to the domains at a page granularity, and Xen tracks both the ownership and use of each page. Each domain has a maximum and a current memory allocation. If the domain is in need for more memory, it may adjust its current memory allocation up to the maximum limit. If a domain wishes to save resources, it may reduce its current memory allocation.

Since Xen allocates memory on a page-level granularity, it cannot guarantee that a domain will receive a contiguous stretch of physical memory. However, most modern operating systems expect memory to be comprised of at most a few large contiguous extents. To help alleviate this, Xen introduces a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to all of the actual hardware memory installed in the machine. The machine memory is comprised of 4kB *machine page frames*, which are numbered consecutively starting from 0. An actual page frame is thus referred to with the same *machine frame number* (mfn) within Xen or any domain.

In contrast, pseudo-physical memory is a per-domain abstraction, providing the guest with the illusion of contiguous memory, even though the underlying machine page frames may be sparsely allocated and laid out in any order. Mapping from pseudo-physical memory to machine memory is the responsibility of the guest OS, which must maintain a *physical-to-machine* table. Xen fascilitates the inverse mapping by providing a *machine-to-physical* table that is readable by all domains.

Memory sharing

Memory can be shared between domains, making communications between both unprivileged and privileged domains possible. The *split device driver* architecture discussed in section 6.1 is actually implemented through the use of shared memory. Memory is shared through the use of *grant tables*. Each domain maintains a grant

table, which is used to tell Xen what kind of permissions other domains are granted on its pages.

6.2.4 Xenstore

Xenstore is a centralized configuration database that is accessible by all domains. It consists of a collection of key-value pairs stored in a hierarchical namespace, and is used to store information about the running domains and as a mechanism for controlling domains. Uses of Xenstore include:

- setting up shared memory regions and event channels
- notifying the guest OS of control events
- reporting status information from the guest OS

Each domain has its own directory hierarchy in the store, containing data related to the domain's configuration. Guests have only write access to the contents of their own directories, but may read any part of the store, but *only* if it has permission to do so. Domain 0 may however, by being a privileged domain, read or write anywhere in the store.

Domains can be notified about changes in subtrees of the store by registering callback functions within the store. The callback function is invoked when anything at or below that point in the hierarchy changes.

6.2.5 Devices

In section 6.1, it was noted that Xen uses a *split device driver* architecture. Figure 6.4 gives a more detailed view of its implementation.

As seen in the figure, the front- and backend layers contain drivers for various types of devices, including network and block devices. The frontend drivers appear as real devices to the guest operating system, but since it does not have access to the physical hardware, all IO requests from the guest kernel are delegated to the backend.

The backend driver is then responsible for ensuring that the requests are well-formed and do not violate isolation guarantees. After validating the requests, the backend then issues them to the real device hardware. By keeping most of the complexity in the backend, the frontend drivers can be kept relatively simple.

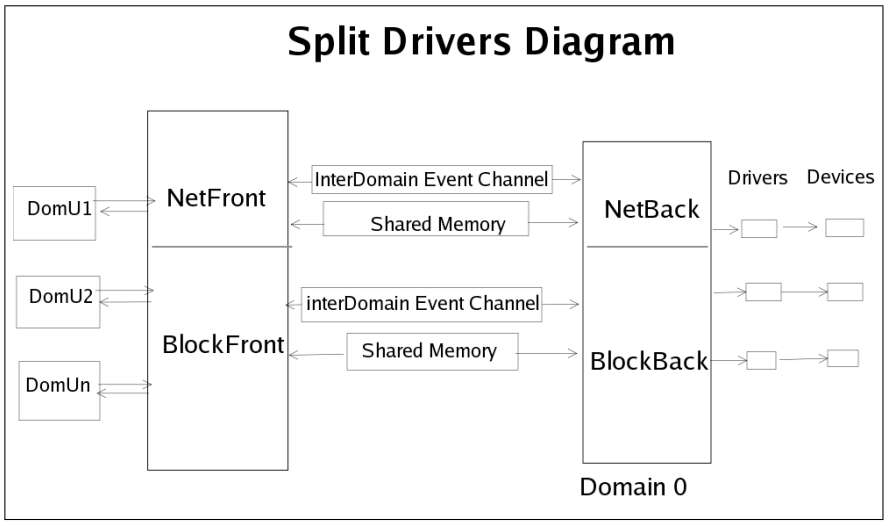

Figure 6.4: Split device driver diagram [62]

After a frontend driver is initialized , an event channel and a shared memory area are set up for communication with the backend. The event channel is used for asynchronous notifications of activity, while the shared memory is used for passing requests and data. The separation of notification from data transfer is done for efficency, by allowing messages to be sent in batches.

6.3 Domain 0

As stated above, Domain 0, the initial domain, has special privileges and responsibilities. Domain 0 is responsible for creating and destroying domains and managing their resources and virtual devices.

By running the control software in domain 0, and not in the hypervisor itself, Xen achieves a separation of mechanism and policy within the system [25].

6.3.1 Xend

Xend is the control daemon used to manage the system running Xen. It runs in domain 0, using a low-level api to communicate with Xen via the domain 0 kernel. Xend exports an HTTP interface to its clients, and commands are issued to xend via the command-line tool *xm*.

6.3.2 xm

The *xm* program is the primary tool for managing Xen through the xend daemon. A complete list of the available commands can be obtained by issuing the command `xm help --long` on the command line; here, only a few examples will be given.

The `xm list` command lists all the running domains. An example output from the command is shown below:

```
Name              ID   Mem VCPUs      State   Time(s)
Domain-0           0   257     1     r-----   7064.6
xenetch           14   128     1     -b----    234.1
```

Name is the name of the domain, while ID is its numeric id. Mem is the size of memory allocated to the domain, while VCPUs it the number of virtual CPUs allocated to the domain. State is the run state, where `r` means that the domain is currently running on a CPU, while `b` means the domain is blocked (waiting on IO, or sleeping because it has nothing to do). Time is the total run time for the domain.

Several commands exist for managing the lifecycle of a domain, including `xm create` for creating a domain, `xm pause` for pausing the execution of a domain, `xm reboot` for rebooting a domain and `xm shutdown` for shutting down a domain.

6.3.3 libxc

For low-level access to the Xen control interfaces, Xen provides a library, *libxc*, written in the C programming language. This library contains not only domain management functions, but also functions for memory mapping, domain debugging and tracing. Chapter 8 will describe the use of this library in retrieving information from the Xen hypervisor.

6.4 Note on performance

The Xen hypervisor has proved to perform close to native Linux on a modified Linux operating system. This result was first published by the Xen developers themselves in the paper *Xen and the Art of Virtualization* [30], and later confirmed by an independent research group in *Xen and the Art of Repeated Research* [35].

Some recent benchmarks, published by VMware and Xensource respectively, can be found in [76] and [78]. These also serve as an example of the many ways benchmarks can be performed, and as an illustration of the importance of *independent* benchmarks.

6.5 Notes on security

The security of Xen is vital for the security of the virtual machines running within it. Assessing the security of Xen involves analysing the hypervisor, the privileged and unprivileged domains, and the ways they interact.

A key assumption is that the hypervisor is likely to be correctly implemented. As Goldberg [39] points out, the limited functionality of hypervisors enables bugs to be largely checked out by running system diagnostics. While this does not prove it secure, it is reasonable to expect a low number of exploitable bugs stemming from the Xen hypervisor.

As described in section 6.3, domain 0 has special privileges and responsibilities. In addition to host the Xen management software, it also functions as the **driver domain**, implementing the backend parts of the split device drivers described in section 6.2.5 [25]. Because domain 0 has effectively unrestricted access to the real device hardware, the security of the other domains is dependent on the security of domain 0.

Also, the limitations in current hardware raises a number of concerns related to the use of hardware device drivers, and the *Xen Users' Manual* identifies at least some of them [26]:

- Hardware devices can make use of DMA (direct memory access) to perform memory reads and writes outside of the memory of its controlling domain. A malicious domain can thus use a hardware device to overwrite or reading memory in another domain.
- Devices that use the same data bus can eavesdrop on each others' data. If the devices are controlled by different domains, one domain could eavesdrop on the data being transmitted to the other domain.
- Xen restricts access to the memory address space on a page size basis. If multiple devices share a page in the I/O memory address space, the domains to which those devices belong will be able to access the I/O memory address space of each others' devices.

- Devices sharing the same interrupt line can block each other from receiving interrupt, or flood the line with interrupts.

However, in a normal setup, only domain 0 will have direct access to the hardware mentioned above.

Part II

Contributions

Chapter 7

Requirements, architecture and design

The previous chapters have provided the background information needed for designing an integrity checking framework based on virtualization. In this chapter, the knowledge obtained is used to formulate a set of requirements that must be fulfilled by the integrity checking framework. This chapter concludes with the description of a general model, serving as a starting point for the implementation of the prototype required by the problem statement given in chapter 1.2.

7.1 Research context

In section 2.4.2, it was shown how Garfinkel and Rosenblum made use of virtualizaton for building an intrusion detection system (IDS) [37]. Their implementation was based on the VMware virtualization system [22], a commercial, proprietary product [1]. The work of Garfinkel and Rosenblum has been of great influence for this thesis, and the lack of publicly available source code both for VMware and the IDS prototype made by Garfinkel and Rosenblum, has only served as an added motivation for providing an open source implementation.

From the Norwegian University of Science and Technology (NTNU), at least two master theses related to kernel integrity checking have emerged during the past few years. One was the thesis of our current daily supervisor Ane Daae Weng [77], and the second was the 2005 thesis of Tobias Melcher [54].

[1] See section 5.4.1 for a description of VMware

In the design of the integrity checking framework presented in this thesis, the authors have been primarily influenced by the work of Garfinkel and Rosenblum, and of the work done by Melcher.

7.2 General requirements

In his thesis Melcher identified some general requirements (GRs) that he, with the support of Weng's [77] work, found to be important for the implementation of an kernel integrity checking system. Table 7.1 lists Melcher's findings.

Requirement	Description
GR1 Kernel malware independence	The kernel integrity model should be able to discover any kind of kernel level malware.
GR2 Integrity of important files	The model should be able to check the integrity of any file within the monitored system.
GR3 Integrity of kernel memory	The kernel integrity model should be able to check the integrity of the kernel's memory.
GR4 Isolation	The kernel integrity checker should be kept strictly apart from the system to be checked.
GR5 Resource consumption	The kernel integrity model should not hurt the performance of the system to be checked.
GR6 Hidden from an attacker	The attacker should be kept unaware of the presence of the kernel integrity checker.
GR7 Operating system independence	The kernel integrity model should be independent of the OS installed on the system to be checked.
GR8 Reporting	The system administrator should be kept aware of any suspicious changes made to the system.
GR9 Reliability	The kernel integrity model should avoid false-positives and false-negatives.

Table 7.1: General requirements of an integrity checking system [54]

The requirements put forward by Melcher provide for an obvious starting point, but a review of each of the requirements is needed before applying them to this thesis. The following will provide a short description of each of the requirements, discussing to which extent they are applicable to the framework presented in this thesis.

GR1 Kernel malware independence The framework should be independent of the type of malware attacking the kernel. The authors' proposed framework

takes this into consideration in the way that it does not care how the kernel is subverted. It looks only at the malware's effect on the kernel's data structures.

GR2 Integrity of important files As the integrity of files falls beyond the scope of the problem definition provided in chapter 1.2, this requirement is not covered in the proposed framework. However, Melcher's work [54] proved that this was indeed doable with a virtualization based integrity system.

GR3 Integrity of kernel memory Melcher's integrity checking prototype was not able to retrieve information from the memory of the guest operating system. As a result, this was an explicitly stated requirement in the problem definition given to the authors.

GR4 Isolation Isolation between the kernel of the virtual machine running the integrity checker and the virtual machine kernel being integrity checked is of utmost importance. This is due to the fact that the host system represents the base of trust for the integrity checking system. Isolation is thus still an important requirement.

GR5 Resource consumption The Xen hypervisor has proved to provide good performance (see 6.4). The proposed prototype will add an extra workload on the processor, by periodically fetching fresh data from the guest domain to analyse. The proposed solution will allow for tuning of the resource consumtion by enabling the administrator to regulate the frequency of this information gathering.

GR6 Hidden from attacker The premise that keeping the integrity checking framework hidden from the attacker is a laudable goal, is debatable. It may be seen as a defense-in-depth measure; by adding the detection of the framework to the attacker's workload. On the other hand, the security of the framework should in the authors' opinion under no circumstances depend on its ability to hide itself.

The integrity checking framework presented in this thesis does not make changes to the monitored system. Since the system cannot be directly observed from the monitored system, its existence can only be deduced from its effect on its environment.

In recent years, the use of virtualization technologies for implementing honeypots [2] has made attackers wary of the technology [42]. Up until recently

[2] A honeypot is an electronic bait, a trap set to be probed, attacked and compromised. The aim of these systems is to collect as much data as possible about the attacker's actions, both during and after an attack.

virtualization has seen very limited adoption in real-world production systems, making the use of virtualization enough to raise an attacker's suspicion. The fact that the virtualization technologies most commonly used incurred a signficant performance penalty, did little to help them pass as regular desktop or server systems.

With the computing industry finding ways to decrease the performance penalty of virtual machines, coupled with the continuing increase of computing power in today's systems, the adoption of virtualization technologies has rocketed in the server market. Xen, the virtualization solution used in this thesis, now claims near-native performance with a "benchmarked overhead of well under 5% in most cases" [27] (for more on Xen's performance, see section 6.4).

The widespread adoption of virtualization, together with its diminishing performance penalties, does much to weaken the correlation between the technology and its use as a detection platform. But even if the presence of an integrity checking system could be deduced from the use of virtualization, the positive effect this would have on the safety of the system would, at best, be questionable. It may even be seen as an incentive for the attacker to move on to a more attractive target.

GR7 Operating system independence The current implementation of the proposed prototype will not be independent of the monitored operating system. The modularity of the prototype will decrese the amount of work required for extending the support for additional operating systems in the prototype.

GR8 Reporting The proposed prototype will be able to report all events to a log file, as well as to take immediate action to pause the suspected running guest domain if something suspicious occurs.

GR9 Reliability The prototype will be able to get a reliable view of the memory areas belonging to the monitored kernel, but the overall reliability of the system will depend on its ability to correctly identify only the malicious changes made to the kernel.

7.3 General model

Figure 7.1, taken from Melchers work [54], show the general idea of an integrity system based on virtualization. In this figure, a guest system is visible to the outside world, and may thus *not* be considered safe from attacks. The host system, on the other hand, is isolated from the guest system with the help of the virtualization monitor, and may be considered safe from attacks from the outside world. Since

both the host and the guest system are running on the same virtualization platform, the host system retains full visibility into the kernel of the guest system, but not vice versa.

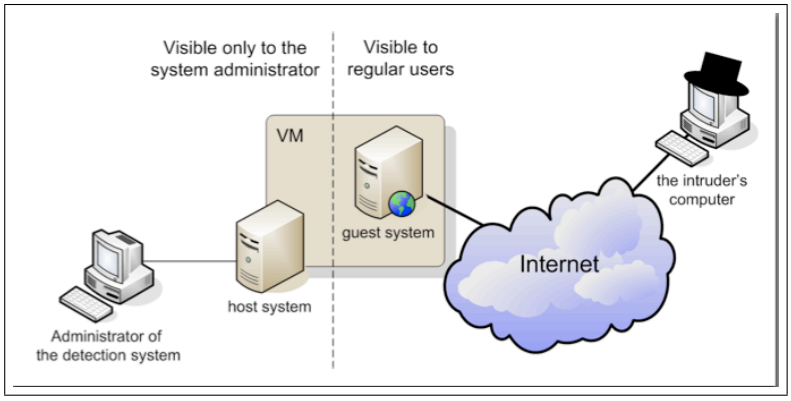

Figure 7.1: General architecture
[54]

Influenced both by the Melcher's model (figure 7.1) and the system designed by Garfinkel and Rosenblum [37], the authors propose the general architecture outlined in figure 7.2. This figure contains the same main parts found in Melcher's figure: The host system, guest systems and the virtualization hypervisor.

7.3.1 The architecture's support for the general requirements

Below is an outline of how the architecture shown in figure 7.2 supports the general requirements of section 7.3.1.

GR1 To support GR1, Kernel malware independence, the proposed architecture introduces Policy modules. These modules define what parts of the kernel's memory are to be checked and what the allowed changes in those parts are. It will be possible to add new modules at run-time, allowing the administrator to select which parts of the kernel structures that should be monitored without having to restart the integrity checking system.

GR2 As mentioned above, GR2, Integrity of important files, will not be covered in this prototype, but it should be relatively straight forward to add support for this by reimplementing the work done by Melcher [54]. His solution to this problem was to mount the guest systems disk(s) in the host system's filesystem and perform

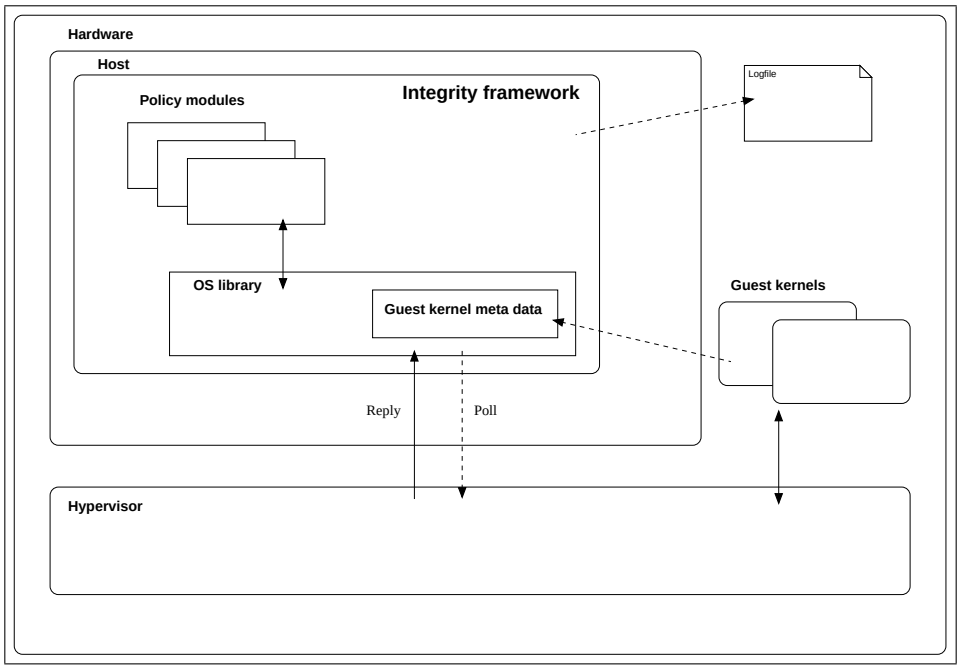

Figure 7.2: General architecture

integrity checking on the mounted disk with the help of a standard file integrity checker.

GR3 Integrity checking the kernel memory requires that the prototype has access to the physical machine memory of the guest system. The physical machine memory is controlled by the virtualization hypervisor, and the hypervisor should therefore be able to provide the prototype with all the necessary information. As can be seen in figure 7.2, the proposed architecture will contain an operating system library (the OS library). This component will be responsible for the communication with the hypervisor and for providing the policy modules with data from the memory of the guest.

GR4 For the isolation between guest and host systems, the architecture relies on the actual implementation of the chosen virtualization solution. Thus, a discussion of the isolation provided in our prototype solution will be required.

GR5 As illustrated in figure 7.2 the communication between the hypervisor and the OS library will be based on a polling scheme. An alternative, but more complex

scheme would be to modify the hypervisor, enabling it to directly notify the prototype when modifications are being made to the monitored memory areas. While the former scheme consumes more resources by constantly pulling, the latter scheme the hypervisor would only alert the OS library when something suspicious happens. By letting the policy modules be able to determine the polling frequencies, the resource consumption imposed may to some extent be controlled.

GR6 GR6 has been rejected as an requirement for the architecture proposed, based on the discussions of section .

GR7 GR7, Operating system independence, is partially covered by the fact that the OS library contains meta data about the guest kernel(s). This library can be adapted or replaced to implement support for different operating systems, as long as they are supported by the hypervisor.

GR8 Reporting will be supported by the logging facility provided by the architecture.

GR9 The reliability of the system will depend both on the virtualization solution's ability to provide the integrity analyser with the correct data, and on the integrity policies abilities to correctly identify only malicious modifications of the monitored kernel.

The next chapter presents an integrity checking framework implementing the architecture presented above.

Chapter 8

Chili - an integrity checking framework

This chapter gives a description of the implementation and inner workings of Chili[1], our framework[2] for integrity checking of a running kernel.

The first section explains the general structure of the Chili framework. The next section gives a presentation of each component of the framework in more detail. Section 8.4 comments on the choice of languages used to implement the framework, and section 8.5 describes the XenAccess open source project that has been used as part of the Chili framework. The last section gives the reader an example of how the framwork can be used.

8.1 Introduction

The framework is implemented as a user mode process running in the privileged domain (see chapter 6). Figure 8.1 shows the overall architecture of the Chili framework. Chili runs in Dom0, the privileged Xen domain, while DomU is the unprivileged domain running the kernel that is to be monitored. The figure also shows some other elements in Dom0 that are used by Chili, but not a part of the framework iteself: The Xen manager and the Libxc are part of Xen, while XenAccess is a third-party library that helps in getting a more convenient access to the run-time state of DomU.

[1] The name "chili" does not mean anything special. It is just a randomly picked codename we used in the development process.

[2] The Chili implementation is called a "framework" as it is easily extended with new functionality through new policies and targets without needing to change the core parts of the system.

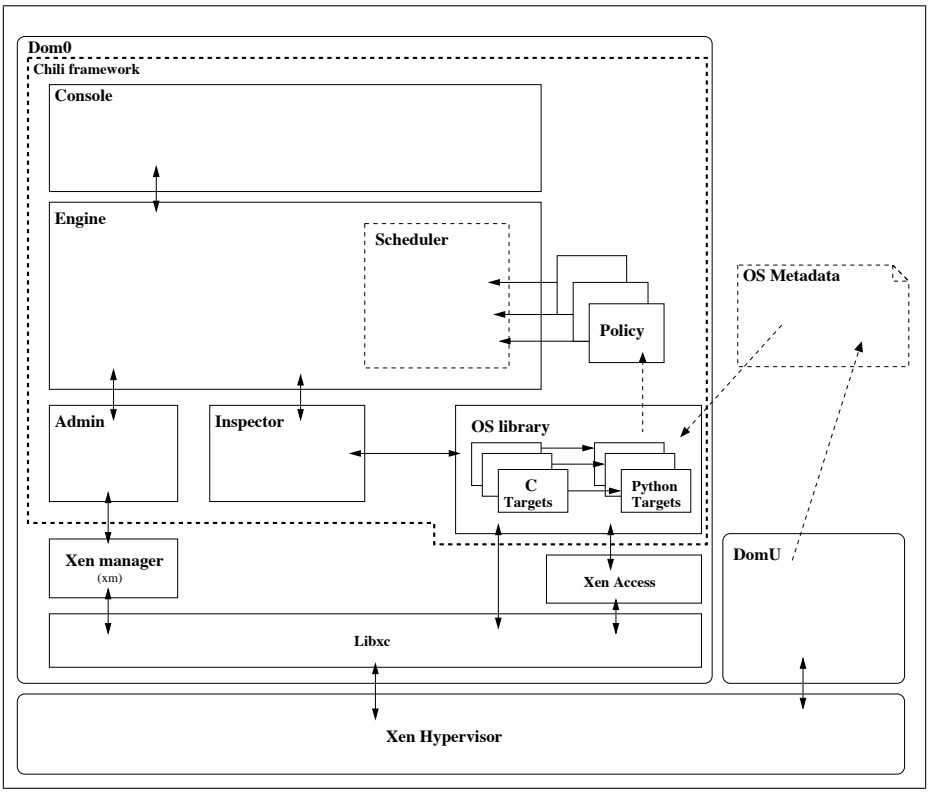

Figure 8.1: Chili architecture

As shown by the figure, the Chili framework consists of several modules. The modules of most importance to the framework are the engine, the policies and the targets. Together these modules make the framework extensible and easy to adopt to the user's needs.

The engine provides the core functionality of the framework. This functionality is however implemented in loosely coupled modules, meaning that the functionality provided by e.g the Admin module (see 8.2.3) or the Scheduler may be replaced without needing to rewrite the engine module.

The policy modules represent the integrity policies that are to be enforced by the framework. They are implemented as plug-in modules, allowing new policies to be tailored to the user's needs and added to the framework at run-time.

The targets, representing the various kernel structures of the monitored domain, are also loosely connected to the rest of the framework. Through the use of predefined interfaces, new targets can be made to collect information about every area of the user domain's memory. The policies and targets will be explained in more

detail later in this chapter.

The rest of this section will give a description of each of the main components composing the framework. All the source code referenced below can be found in appendix E.

8.2 Description of the Chili framework

This section gives a description of all the modules shown in figure 7.2.

8.2.1 Console

The Chili framework is controlled through a command line interface, called the Console. The console communicates only with the engine, functioning as a messenger that brings user requests to the engine. Section 8.6 gives a more detailed description of the use of the Chili Console, and its source code can be found in Appendix E.1.

8.2.2 Engine

This is the heart of the framework, implementing most of the critical functionality of the framework. The engine is responsible for loading all available policies at startup, enabling them for later activation. It also sets up a logging mechanism which makes it possible for the policies to write their output to a log file, named `chili.log`.

One of the main components within the Engine is a multi-threaded scheduler. Upon the activation of a policy, the Engine registers the policy with the scheduler, which causes the policy to be run at a periodic interval. Rather than implementing a full-fledged multi-threaded scheduler of our own, the scheduler of the *Webware for Python* web application toolkit has been integrated into Chili.

The Engine also makes use of the Admin and Inspector modules, both of which will be described shortly. The source code implementing the Engine is found in Appendix E.2.

8.2.3 Admin

The Admin module is used by the Engine to send administrative commands to the guest domains. This module basically just wraps the `xm` tool (see section 6.3.2) which is provided as part of the Xen installation. The current implemented methods are: `listDomains`, which lists all the current running Xen domains,

`pauseDomain`, which temporarily halts the execution in of a given domain and `unpauseDomain`, which resumes the execution of a given domain. Its source code can be found in appendix E.3.

8.2.4 Targets

The targets represent the different kernel structures of kernel running in the monitored domain. The targets are specially tailored for each type of kernel stucture, reflecting the semantics of the structure. One exception is the generic target `MemoryArea` described more fully in section 9.1.6, which can represent any memory region defined by a *start* and *end* virtual address.

The targets are implemented using the C language. The implementation makes heavy use of both the Xen control library: *libxc* (described in section 6.3.3) and the *XenAccess* library (described in section 8.5). All C targets are also provided with a Python wrapper, making them easier to integrate with the rest of the framework (see section 8.4 for the details).

The authors have written several targets to retrieve information from the monitored kernel. For detailed explanations of these, please see chapter 9 and the source code of Appendix E. To add support for other kernel structures, new targets will have to be added to the framework. This can be easily done without requiring changes to be made to the surrounding framework.

8.2.5 OS library

The OS library serves as a wrapper around the framework's available targets. It is responsible for instantiating and returning a fresh target when requested by the Inspector. Every time the OS library gets a request for a specific target, it creates a new instance of this target, reflecting the state of the corresponding kernel structure at that time.

The name *OS library* indicates that the framework may support multiple operating systems. As this is the only place in the framework that is exposed to the details of the implementation of the monitored kernel, it would be the right place to implement support for different operating systems. This could probably best be done through differentiated implementations of the framework's targets. In this way one could have one implementation of e.g the kernel text target (see section 9.1.1) for the RedHat Linux distribution, but a completely different one for Windows XP. The rest of the framework would not have to be changed to support multiple operating systems.

8.2.6 Inspector

The Inspector serves as the link between the Engine and the OS library, and the policies running inside the Engine uses the Inspector to request fresh targets from the OS library.

Because the Inspector provides no functionality of its own, the reader may wonder if a cleaner and easier solution would have been to let the policies call the OS library directly. A decoupling between the policies and the OS library was however desired, and the reason can be found in figure 8.2 which shows the framework with an additional module, the Monitor. In the framework's current form, the policies themselves monitors the kernel structures for changes, by periodically requesting new targets from the Inspector and comparing them with the previous ones. This is not a very efficient solution, particularly if the targets in question are used in multiple policies. As a result, the concept of a separate Monitor module was introduced. The policies would register their interest of a particular target type with the Monitor, which would notify all interested policies when the target changes. The policies would still be able to inspect other kernel structures using the Inspector, but the actual monitoring of kernel structures would be best left to the Monitor. The Monitor could also be extended to make use of the event mechanisms provided by Xen through the use of Xen store, allowing for more efficient monitoring of certain events.

Monitors were not implemented in the current prototype due to the limited amount of time available, but the concept may prove useful if the prototype is to be expanded.

The source code for the Inspector module can be found in appendix E.5.

8.3 Policies

For the framework to be of any real use, there has to exist a set of rules that define which parts of the monitored domain's kernel structures that are allowed to be modified and which that are not. The policy modules define these rules. Each policy contains a set of rules based on the policy's knowledge about one or more of the targets available in the framework. The distinction between a target and a policy is worth repeating: The target represents the specifics of a kernel structure. The policy defines both what is considered to be illegal changes in the kernel structure, and what the consequences should be.

As described in the preceding section, the monitoring of the kernel structures is currently done by the policies themselves. When a policy is first activated, it

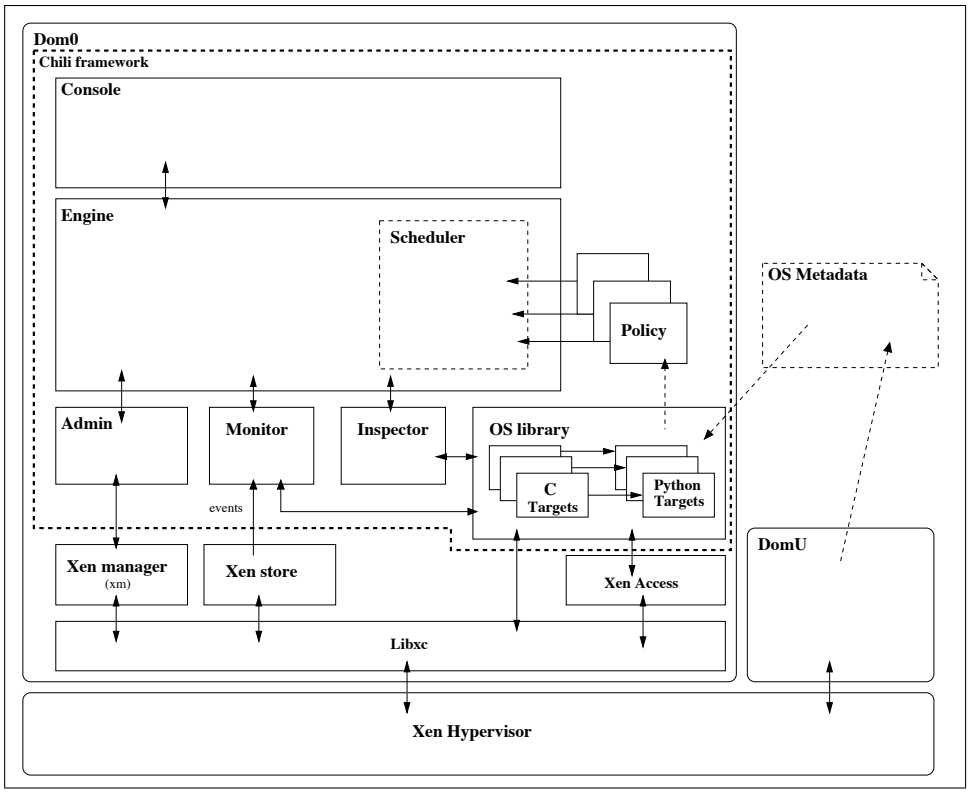

Figure 8.2: Chili architecture with monitor

requests a copy of the target(s) of interest. In regular intervals, a new copy of the target is requested and subsequently compared to the original one, thus detecting if the target has changed.

The authors have written several policies for use in the Chili framework. For detailed explanations of these, please see chapter 9. The source code of the policies can be found in Appendix E.

8.4 Choice of language

The framework is written mostly in the Python language [14]. Some of the more low-level parts are written in the C language. Python were chosen as the main language of the implementation because it is a powerful, high-level object-oriented language that offers strong support for integration with other languages. Python is also the language of choice for the high-level parts of the Xen hypervisor, and this makes it easy to integrate the chili framework with Xen (see section 8.2.3).

Some parts of the code were however found to be more convenient implementing in the C-language, as it allows for low-level pointer arithmetics to be performed. To make the C-parts of the code integrate smoothly with the rest of the framework, each C-module was wrapped in Python code (see below).

8.4.1 SWIG

The wrapping of the C code was done by utilizing the Simplified Wrapper and Interface Generator (SWIG) tool [19]. This tool uses special files called *interface files* to describe how the Python wrapper codetool should be generated. SWIG does the work of creating the relatively complex Python module required to seamlessly use the functionality defined in the C files. The generated Python module can be used like any regular Python module, thereby making the methods and data structures of the original C file available in a Python-native way.

8.5 XenAccess

The *XenAccess* library is an open source project trying to simplify the way in which the hardware state of guest domains can be viewed by a privileged domain. The project is still in its early stages, but the authors found that it greatly helped to raise the abstraction level when accessing machine memory using the virtual addresses of kernel of the guest domain. The project is frequently adding new features, making it likely to be of great help in a future version of the Chili prototype supporting multiple operating systems. The *XenAccess* library is used extensively by the targets described in section 8.2.4. *XenAccess* is written purely in C, and this made C the natural choice of language for the targets as well.

8.5.1 Limitiations of XenAccess

While the *XenAccess* library was of great help, its support for the memory mapping of areas spanning multiple page frames was somewhat limited. The memory paradigms employed by Xen (see 6.2.3) necessitates each and every page frame associated with a virtual address range to be looked up. As the functions provided by the *XenAccess* library only map a single page frame per invocation, the overhead of mapping areas spanning multiple frames would quickly add up. To solve this, some convenience functions for effectively mapping multiple page frames were developed using both *XenAccess* and the Xen C library itself. These functions were gathered in the library *libca*, and the source code can be viewed in Appendix E.36. An example of the use of this library can be found in the C implementation of the *kernel text target* (see Appendix E.29).

8.6 Use of the chili framework

This section shows the general use of the Chili framework. The chili framework runs on a privileged domain (dom0) while monitoring the kernel in a user domain (domU). The framework is controlled through a text based interface in a console window (see figure 8.3). The available commands are:

setdom Tells Chili which domain to monitor. The domain ID (see chapter 6) of the desired domain is given as an argument.

listPolicies Lists all available policies. Given the option *active*, this command lists only the policies that have been activated.

listTargets Lists all the targets that are registered within the framework.

listDomains Lists the status of all domains currently running in Xen.

activate Activates a policy. The input to this command is either the name of the policy to be activated, or the special keyword “all”, meaning all of the available policies. When activated, the policies are registered with the scheduler. The sensitivity level of a policy can be set upon activation by supplying the parameter “LOW” or “HIGH”

deactivate Deactivates a policy. The input to this command is either the name of the policy to be deactivated, or the special keyword “all”, meaning all of the active policies. Upon deactivation, the policies are removed from the scheduler, causing the monitoring activities started by that policy to stop.

unpause Unpauses a domain. The domain ID of the desired domain is given as an argument. The unpause command is of use when a policy has paused a domain due to a policy breach.

setLogLevel Set the log level of Chili. If given the argument “file”, Chili will only record events in the log file. Given the argument “stdout”, Chili will output the log both to the log file and the console window.

help Lists all the available commands. If the help command is given a command name as parameter, it prints the help string of that command.

quit Quits Chili.

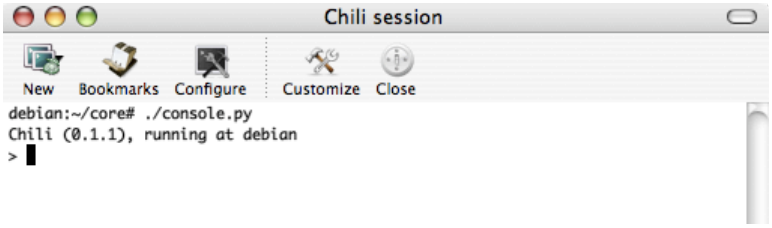

Figure 8.3: Chili initialization

8.6.1 Initialization

The Chili prototype and its dependencies are all initialized when the console starts up, as shown in figure 8.3:

If the initial start up command (`./console.py`) is given a number as its first argument, the framework registers the user domain with the corresponding domain ID as the current working domain. If "-v" or "-verbose" is given as a parameter, the console starts in verbose mode, printing all log info to the console window also. If "-a" or "–activateAll" is given as a parameter, the framework starts with all available policies activated. Figure 8.4 shows the initialization process with all the optional parameters given.

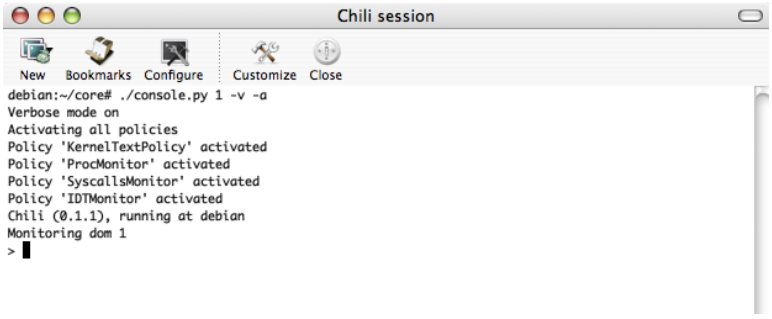

Figure 8.4: Chili initialization

8.6.2 Setting the user domain

To make chili aware of which user domain the activated policies should monitor, the domain id of the user domain must be set in chili. This can be done with the command `setdom` as shown in figure 8.5 below or as described in section 8.6.1.

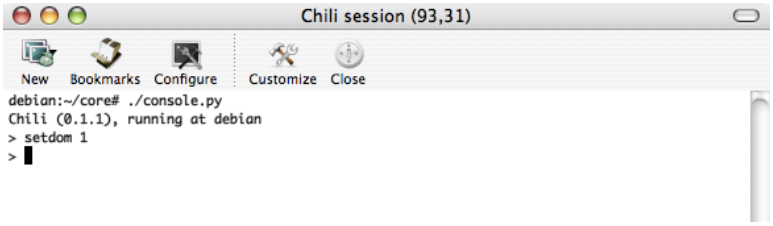

Figure 8.5: Setting the active user domain

8.6.3 List domains

Figure 8.6 shows how to get a list of the currently running domains with the `listDomains` command.

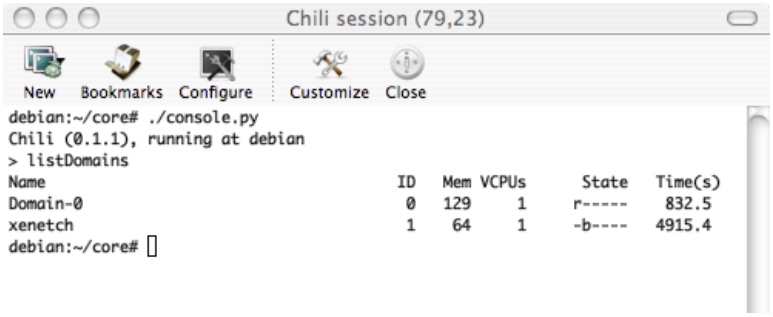

Figure 8.6: Chili listing domains

8.6.4 Targets

To see what targets the framework supports, the command `listTargets` can be used, as shown in figure 8.7, where the framework lists the targets *Proc, Syscalls, IDT* and *KernelText* as registered with the framework.

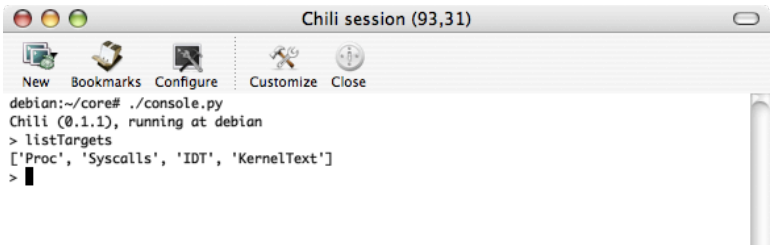

Figure 8.7: List of targets

8.6.5 Policies

The command `listPolicies` lists the policies that are available. If given the option `active`, it shows only the active policies.

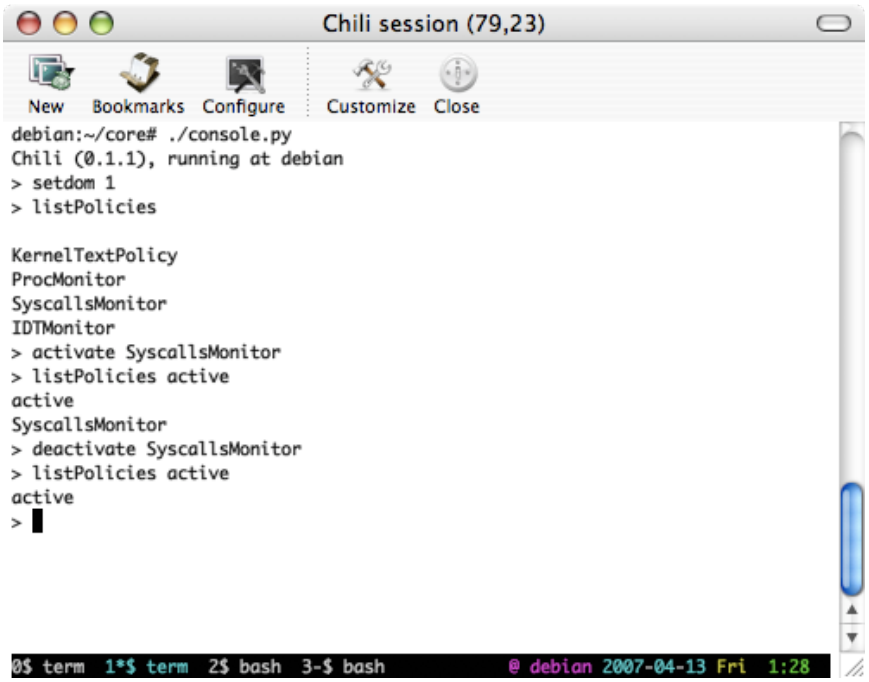

Figure 8.8: List of policies and (de)activation of policy

Figure 8.8 shows that *KernelTextPolicy, ProcMonitor, SyscallsMonitor* and *IDTMonitor* are loaded by the framework. The figure also shows how to activate a policy with the `activate` command, as well as how to deactivate a policy with the `deactivate` command.

Unpausing paused domain

When a domain has been paused by Chili, it can be unpaused with the command `unpause <domID>`, as can be seen in figure 8.9.

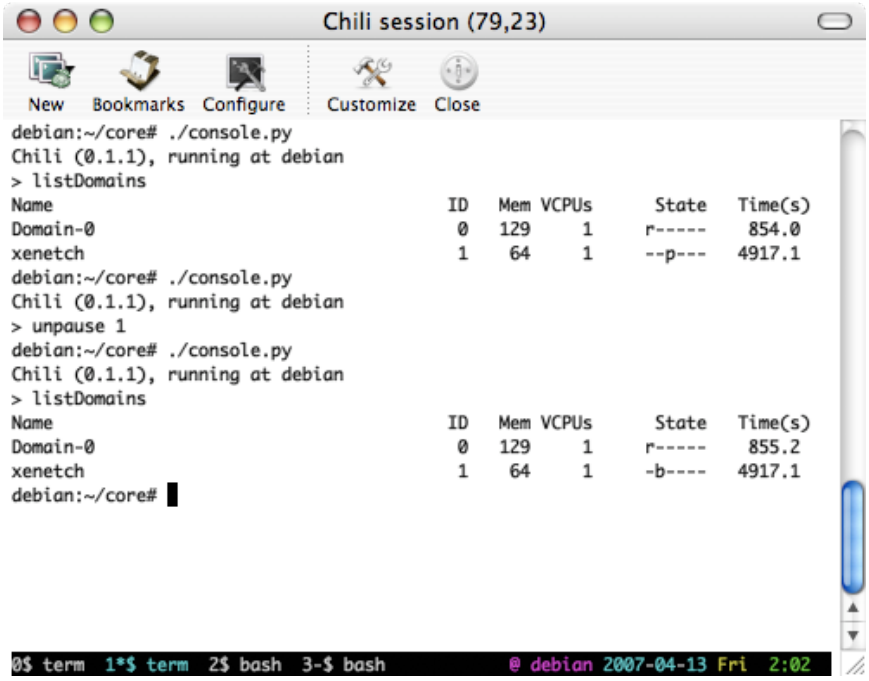

Figure 8.9: Un-pausing a paused domain

Chapter 9

Constructing targets and policies

This chapter describes the example targets and policies constructed for use with the Chili framework. Section 9.1 presents the kernel structures chosen as example targets for integrity checking, while section 9.2 presents the policies making use of the targets.

9.1 Example targets

In the terminology used in Chili, the different parts of the kernel to be monitored are called targets (see section 8.6.4). This section describes the different targets currently implemented and available for use by the policy modules. Each target were developed to monitor one or more internal kernel structures in the guest system's memory. The choice of data structures was motivated by the attack techniques used by today's rootkits (described in chapter 4).

The reader is encouraged to consult the source code listed in appendix E to get a more detailed understanding of each target. When referring to the source code files, their location within the appendix will be given in the parenthesis following them.

9.1.1 KernelText

Figure 3.2 on page 21 shows the kernel as it is represented in memory. The area in the figure marked as *kernel code* is commonly referred to as the text part of the kernel. By modifying the kernel text, malicious code may add its own hooks to common kernel functions.

The kernel text is represented by the *kernel text target* in Chili. This target stores a checksum computation of the kernel code, enabling the detection of any changes made to it.

The target is implemented in the python module *KernelText.py* (E.28), and in the file *kernel_text_target_data.c* (E.29). The SWIG interface file can be found in the file *kernel_text_target_data.i* (E.31).

9.1.2 SysCallTable

A popular attack point amongst malware authors is the *system call table* described in section 3.3.6. As depicted in figure 3.4 on page 23, the system call table contains function pointers to the service routines implementing the system calls.

A target representing the system call table has been constructed, allowing for the detection of changes made to the table. The target stores a checksum of the entire table, together with the individual entries comprising the table. It is thus able to determine which entries that have changed.

The python implementation of this target can be found in the file *SysCallTable.py* (E.12), and the C-implementation in the file *sct_target_data.c* (E.13). The SWIG interface file is in the file *sct_target_data.i* (E.15).

9.1.3 ServiceRoutineArray

A target representing the service routines that implement the system calls was also created. It contains a checksum for each of the service routines, as well as a single checksum computed from the individual checksums, thus allowing for quick comparison between instances of this target.

The current implementation of this target is far from ideal as it just checks the first 31 bytes of each service routine. In a non-prototype version of this target this is not sufficient as it will not detect modifications after the 31st byte. A more complete implementation of this target would have to be given information about the length of each of the service routines, making digests based on those.

The python implementation of this target can be found in the file *ServiceRoutineArray.py* (E.16), and the C-implementation in the file *sr_target_data.c* (E.17). The SWIG interface file is in the file *sr_target_data.i* (E.19).

9.1.4 IDT

As described in section 3.3.5, the interrupt descriptor table contains the mappings between the various interrupts and the functions that handle them. A target representing the IDT was therefore deemed useful.

In a non-virtualized Linux system, the address of the IDT resides in the `idtr` CPU register. Xen provides each domain with a virtual IDT, which can be accessed and modified through hypercalls. Malicious code would thus need to be Xen-aware when trying to modify the IDT.

The entries in the virtual IDT are data structures containing information about each trap vector (see the file *idt_target_data.h* E.22). The implementation of this target maintains an object representing all of the entries in the IDT, as well as a checksum computed from the entire IDT. The Python and C implementations can be found in the files *IDT.py* (E.20) and *idt_target_data.c* (E.21) respectively. The SWIG interface file is in *idt_target_data.i* (E.23).

9.1.5 Proc

The */proc* file system described in section 3.4.2 provides information about the processes running on the system. Being able to check the integrity of the */proc* file system is thus highly desirable, and a target enabling the monitoring of some of the key parts of the */proc* file system has been created. The structures containing the function pointers to the methods that the kernel can invoke in the */proc* file system are represented in the target. In addition, the target contains a digest of these function pointers.

The implementation can be found in the files *Proc.py* (E.24) and *proc_target_data.c* (E.25). The data structures can be found in *proc_target_data.h* (E.26). The SWIG interface is in the file *proc_target_data.i* listed in E.27.

9.1.6 MemoryArea

A generic target capable of representing any area of kernel memory was also developed. By providing this target with either a memory address or a kernel symbol, together with the size of the area to be represented, a checksum of the memory area is computed. While the MemoryArea is flexible in the way that it can be used to represent any memory area within the kernel, the representation of the area is limited to a checkum.

The implementation can be found in the files *MemoryArea.py* (E.32) and *memory_area_target_data.c* (E.33). The data structures can be found in *memory_area-*

_target_data.h (E.34). The SWIG interface is in the file *memory_area_target_data.i* listed in E.35.

9.2 Example policy modules

The mechanisms provided by the Chili framework made several approaches to integrity analysis viable. Rather than opting for the familiar *signature detection* approach used by virus scanners, an approach based on *run-time code attestation* (as described in section 2.4.1) was chosen in this thesis. Directly monitoring the integrity of important kernel segments, allows their integrity to be validated against both known and unknown threats.

The currently implemented policies handle policy breaches in two different ways, depending on the sensitivity set in the policy. If the sensitivity is set to "HIGH" the module pauses the affected user domain. If the sensitivity is set to "LOW", the event is just noted in the log file and the user domain is allowed to continue its execution.

9.2.1 The __kernel_vsyscall policy

As shown in section 3.3.6, the `__kernel_vsyscall` function is called by the standard library when invoking a system call, causing the execution of either the `int $0x80` or the `sysenter` assembly language instructions. By modifying the `__kernel_vsyscall` function, an attacker may be able to hijack the system call invocation.

This policy module is able to detect modifications of the `__kernel_vsyscall` function. It uses the generic MemoryArea target described in 9.1.6 to obtain a digest of the `__kernel_vsyscall` function and compare it to a previously known good state. The implementation of this policy can be found in *KernelVsyscall.py* (E.10).

9.2.2 The IDT policy

As shown in section 4.4.2, some kernel-level malware modify the entry in the IDT that contains the address for the *system_call_function* (this technique is for example used by the SucKIT rootkit (see appendix D.2)). This policy module is able to detect this kind of modifications of the IDT. It uses the IDT target, described in section 9.1.4, to get information about the state of the IDT and to compare this with its previously known good state. The implementation can be found in *IDTMonitor.py* (E.7).

9.2.3 The kernel text policy

This policy monitors the static text part of the kernel code by regulary polling the kernel text target described in section 9.1.1. The implementation of the kernel text policy can be found in the file *KernelText.py* (E.28).

9.2.4 The system call policy

As described in chapter 4.4.2, manipulating system calls is a common technique used by malicious code. This policy monitors not only the system call table for changes, but also the service routines implementing the system calls.

The `sys_call_table`, which contains the mapping between the system calls and the service routines implementing them, is represented by the SysCallTable target. The service routines are represented by the ServiceRoutineArray target. The implementation of the system call policy can be found in the file *SyscallMonitor.py* (E.8).

9.2.5 The proc policy

A policy module has also been written to ensure the integrity of the */proc* file system, using the Proc target. The implementation is listed in *ProcMonitor.py* E.9).

9.3 System calls - a case study

The targets and policies developed were chosen based on the kernel functionality they implement. For example, the system call mechanism of the kernel was identified at an early stage as being crucial for the integrity of the kernel. The system call mechanism is a favorite target of malware authors, and it can be attacked at various places.

The following will use the system call mechanism, all the way from the invocation of a system call to the service routine implementing the system call, as an illustration of the complexity faced when trying to verify the integrity of a given kernel functionality. Figure 9.1 shows both the call chain of the system call process, as well as the policies used to protect it.

In section 3.3.6, it was described how system calls can be invoked by using either the `int $0x80` assembly instruction or the `sysenter` assembly instruction. The wrapper functions in the *libc* standard library make use of the `sysenter` instruction if this is supported by the CPU. This is made transparent for the libc library by letting its routines call the `__kernel_vsyscall` function, which in

turn invokes either the `int $0x80` or the `sysenter` instruction, depending on the capabilities of the CPU. As the `__kernel_vsyscall` function presents itself as a target for hijacking the system call process, the __kernel_vsyscall policy was created for protecting it.

The IDT, which contains the mapping between the `int $0x80` assembly instruction and the `system_call` function, is protected by the IDT policy.

The code for both the `system_call` and the `sysenter_entry` functions is contained within the text segment of the kernel, and are thus protected by the kernel text policy.

Both `sys_call_table`, mapping the system calls to the service routines implementing them, and the service routines themselves are covered by the system call policy. While the system call table are fully protected, the service routines are only partially protected, as dicussed in section 9.1.3.

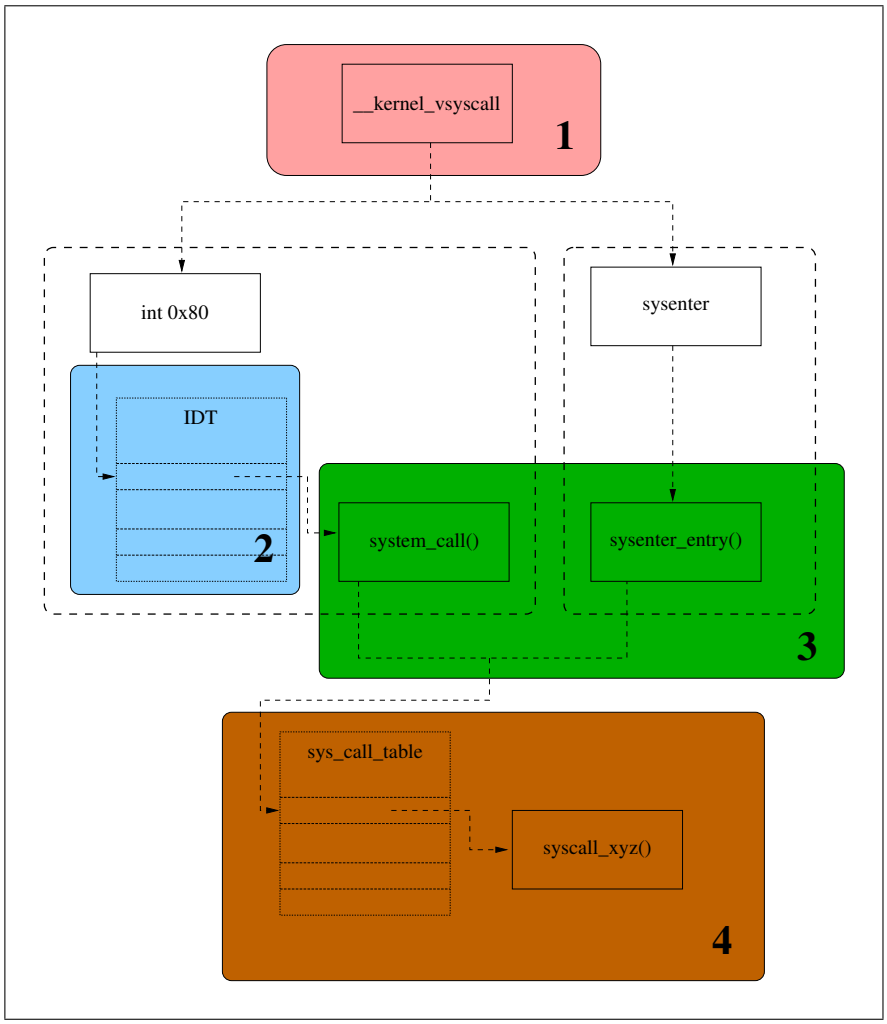

Figure 9.1: The system call mechanism and the policies protecting it. The policies are: (1) the __kernel_vsyscall policy, (2) the IDT policy, (3) the kernel text policy and (4) the system call policy.

Chapter 10

Testing the framework

This chapter presents the testing of the integrity policies described in chapter 9. The tests explore the effectiveness of the Chili prototype and of the example policies and targets, against some common attack methods. This was done to ensure that the framework with example policies and example targets would work in practice.

Our test system was composed of a Pentium IV 2.0 GHz computer with 512 MB of memory. The virtual machine monitor was an unmodified Xen hypervisor, version 3.0.4. The set up and installation of the Xen system are described in appendix C. The privileged domain was running the stock XenLinux 2.6.16 kernel on top of Debian Etch. The user domain, which kernel was to be deliberately subverted, was running the same kernel.

The suite of sample attacks was chosen from some of the most popular rootkits running on the Linux 2.6 kernel. These were: Adore-ng, Override, eNYeLKM and Mood-NT. Additionally, a self-made rootkit, named Chilirootkit, was used. Chilirootkit is a loadable kernel module written by the authors to better understand the way rootkits operate.

The rootkits are described in appendix D, and the implementation of the *Chilirootkit* is outlined in Appendix A.

Section 10.1 shows how the Chili framework was set up prior to the testing, while section 10.2 describes the actual tests. A summary of the test results is given in section 10.3.

10.1 Preparing the Chili framework

Before each test, the chili framework, with the previously described example targets and policies, was initialized from the command line with the command: `console.py 1 -v -a`. The first argument gives the domain ID of the guest domain running on the Xen hypervisor. The `-v` option puts the chili framework in verbose mode so that everything that happens gets printed to both the console and the log. The `-a` option activates all available policy modules right away. This can be seen in figure 10.1. This figure also shows the currently running domains.

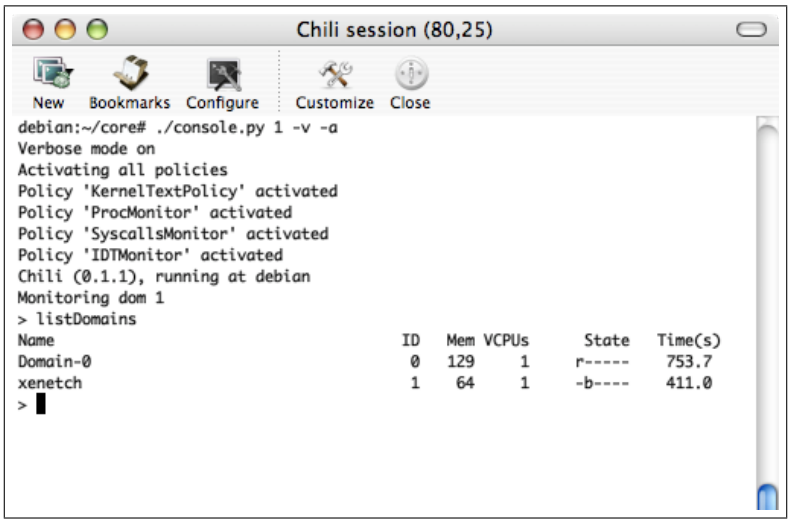

Figure 10.1: Initialization of Chili

10.2 Test results

10.2.1 Chilirootkit

The Chilirootkit was made as part of the learning process while studying the techniques used by rootkits. It is implemented as a kernel module, capable of modifying the system call table. More specifically, it replaces the service function for the *sys_nanosleep* system call with its own malicious version (see Appendix A for details).

Changing the system call table is a very common form of attack. Of the 12 rootkits used in the paper *Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor* [57], 10 of them used this technique (see figure 4.1). This is therefore a very significant test of the practical capabilities of the system.

After initializing the Chili framework as described in section 10.1, the Chilirootkit was inserted into the guest kernel by issuing the command `insmod chili_rootkit.ko`. The output from the guest system's console can be viewed in figure 10.2.

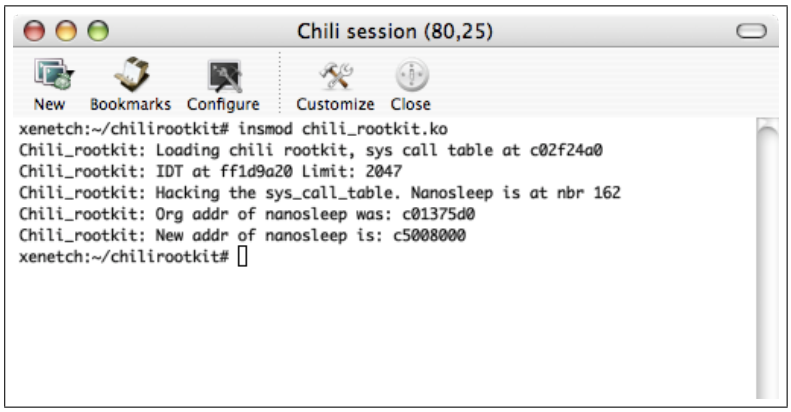

Figure 10.2: Guest domain loading the chilirootkit

Figure 10.3 shows how Chili reacted to the insertion of the Chilirootkit. The change to the system call table was detected, and both the original and current addresses of the the sys_nanosleep function were displayed.

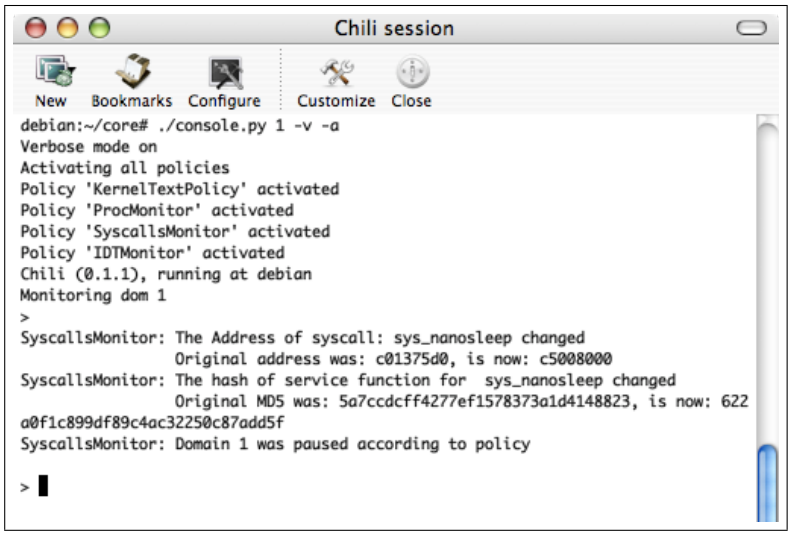

Figure 10.3: Chili responding to chilirootkit

The integrity system reports that it has paused the affected domain. This can be verified by running the `listDomains` command. The "p" on domain 1 indicates

that the domain is currently paused. The domain can then be un-paused by issuing the command `unpause 1`. This can bee seen in figure 10.4.

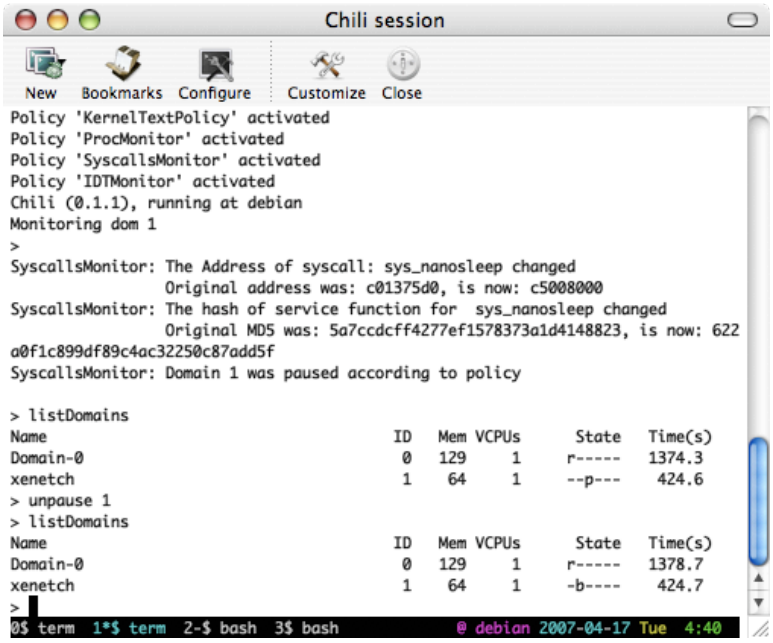

Figure 10.4: Chili un-pausing paused domain

10.2.2 Adore-ng

Next, Chili was tested against the Adore-ng rootkit. This rootkit targets, amongst other things, the Proc virtual file system (see section 3.4). Adore-ng uses this to be able to hide its existence without using the more common technique of overwriting system call jumps.

Adore-ng was installed by issuing the command `insmod adore-ng-2.6-.ko` in the guest domain. As expected, Chili responded almost immediately with a message from the ProcMonitor policy module saying that the kernel was subverted. The output can be viewed in figure 10.5.

10.2.3 Override

The third scenario used the rootkit *Override* [13], which employs the technique of system call interception described in section 4.4.2. Override's own functions act as wrappers for the original ones, filtering the output to avoid detection of certain files and processes.

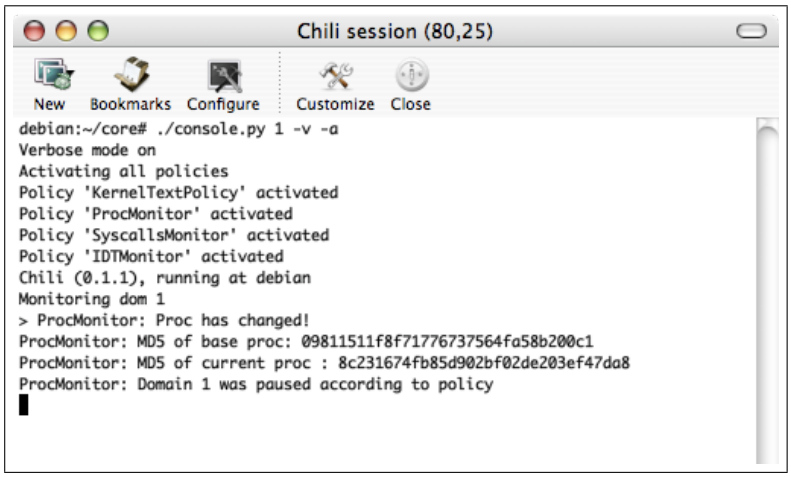

Figure 10.5: Chili reacting to Adore-ng

The Override rootkit would not compile without some slight modifications to its source code. For the sake of completeness, the diff between the patched and the original version is found in Appendix E, listing E.36.3. While Override did not crash when it was inserted into the kernel, it however crashed upon user interaction with the guest os' console.

The SyscallMonitor policy detected five changes made to the system call table. The addresses of the system calls `sys_read`, `sys_chdir`, `sys_getuid16`, `sys_geteuid16` and `sys_getdents64` all changed, as shown in figure 10.6. After inspecting the source code of Override, it was indeed confirmed that these five system calls are the ones intercepted by Override (see appendix D.4).

10.2.4 eNYeLKM

The next rootkit of the testing suite was eNYeLKM [6], another LKM rootkit. This rootkit does not modify the system call table nor the interrupt descriptor table, but instead modifies the system_call and sysenter_entry handlers. The __kernel_vsyscall policy should be able to detect this. Unfortunately the rootkit caused several oopses[1] in the guest kernel when inserted, and could not be made to work properly. No results were therefore obtained from this test.

[1] An oops occurs when some programming defect or otherwise unexpected event interferes with the normal operation of the Linux kernel. It is named for the error message displayed on the system console (or seen in the system log files) When this happens the currently active task (or process) is terminated and the kernel makes a best-effort attempt to continue operation. [23]

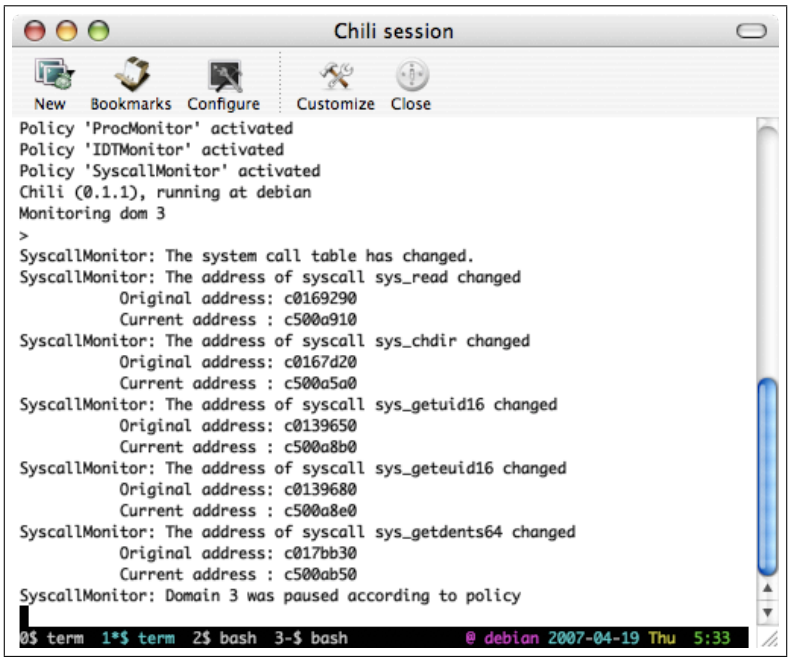

Figure 10.6: Chili reacting to Override

10.2.5 mood-nt

In the last test, the *mood-nt* rootkit was used. mood-nt patches the kernel through /dev/kmem, as described in section 4.4.2. Similiar to override, mood-nt makes modifications to the system call table.

Running mood-nt also caused several kernel oopses in the guest kernel, and the mood-nt binary crashed with a segmentation fault. Before the crash of mood-nt, Chili did however detect a change to the system call table, as shown in figure 10.7.

10.3 Summary

Of the five rootkits in our test suite, four were run succesfully. Chili was able to detect all four, due to their modifications of key kernel structures. It is worth noting that, with the exception of adore-ng, the rootkits all modified the system call table. As adore-ng modified the proc file system, the only two targets triggering detection were the Proc target and the SysCallTable target.

While testing the Override rootkit, some interesting observations were made, resulting in several changes to the SyscallMonitor policy and its targets. In an early version of the SyscallMonitor policy, each system call was checksummed by fol-

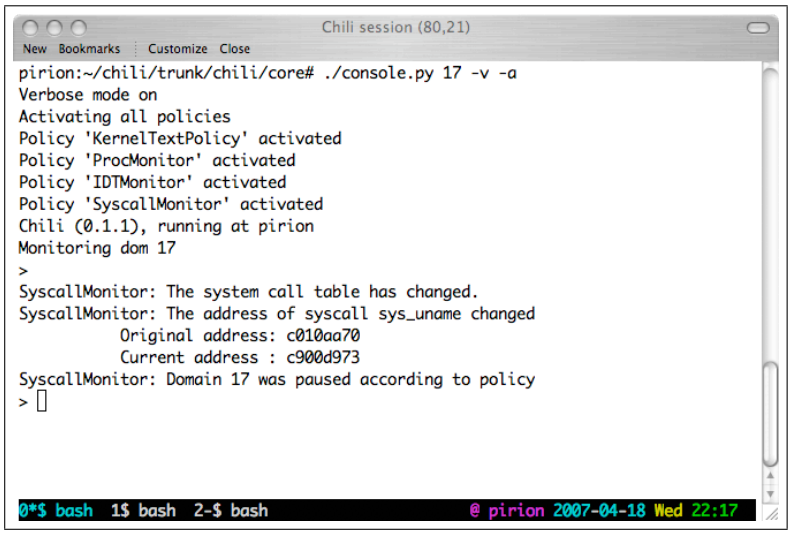

Figure 10.7: Chili reacting to mood-nt

lowing the function pointers of the system call table. As described above, Override substitutes five of the system call table entries with pointers to its own functions. However, due to Override not running correctly in Xen, following the pointers would result in segmentation faults, causing Chili to crash. Although not an intended feature of Override, the incident clearly demonstrated that great caution is required when dealing with the data acquired from the guest operating system. When treating data from the guest as input for Chili functions, sanity checks must be applied.

Chapter 11

Discussion and evaluation

The results from chapter 10 show that virtualization is well-suited as a platform for performing integrity analysis. Based on the test results and available literature, this chapter will discuss the advantages and the limitations when using virtualization as a platform for integrity checking. The first section will focus on the use of virtualization as a platform for doing integrity analysis, while the second section will look at the challenges still remaining in the analysis of the integrity of a running kernel.

11.1 Virtualization as a platform for integrity analysis

This section will discuss the suitability of virtualization as a platform for doing integrity analysis. The platform will be assessed with regards to its visibility, isolation and performance properties. In addition, possible vulnerabilities of the platform will be discussed.

11.1.1 Visibility

The use of virtualization allows for the direct inspection of both memory and CPU state. All the parts comprising the kernel, whether code or data, static or dynamic, are thus made available to the integrity analyser. This ensures the *completeness* of the data set used for the analysis. In addition, the *trustworthiness* of the data set is greatly increased, because the acquisition of data is no longer reliant on the subverted kernel.

11.1.2 Isolation

Keeping the integrity analyser separated from the monitored system is important for two reasons: keeping the analyser's view of the monitored host independent of the host itself and keeping the analyser safe from attacks. While the former is accomplished through the visibility property already discussed, the latter is the topic for this section.

Isolation provided by Xen

Xen provides a high degree of isolation between the virtual machines. However, as the virtual machines are all running on the same physical hardware, full isolation is difficult to achieve. In section 6.5, it was pointed out how the isolation property could possibly be threatened due to limitations in current hardware devices. This presupposes that the virtual machine is given direct access to the hardware device in question, which would not be the case for the monitored host.

As described in chapter 6, Xen provides mechanisms for the sharing of memory between domains. The split device driver model of Xen is implemented using this mechanism. While the use of memory sharing makes for very efficient device access, it severely weakens the isolation between domains.

11.1.3 Performance

The performance costs of the integrity analysing framework presented in this thesis are comprised of the overhead caused by the virtualization solution and the costs of doing integrity measurements and analysis.

Virtualization costs

As noted in section 6.4, running a paravirtualized Linux on Xen is claimed to offer near-native performance. Benchmarking of virtualization solutions is however a complex process, and the agendas of those performing the benchmarks should always be taken into account. However, most of the benchmarks referred to in section 6.4 suggests that the paravirtualization approach of Xen provides significant performance benefits. While the exact numbers are debatable, there is little doubt that the performance penalty related to virtualization has been steadily decreasing the last few years.

The costs of integrity analysis

Although the performance costs imposed by the Chili prototype have not been measured, some reflections on the subject can still be made.

The integrity measurements taken, only involve memory reads. Accessing memory is a cheap operation, especially when compared to accessing disk storage media. The checksumming function uses an existing C-implementation of the highly efficient MD5 algorithm. The surrounding framework, which manages and analyses the integrity measurements, is written in Python, a high level language prioritizing readibiliy over speed. Being a prototype, the Chili implementation is far from optimized with regards to efficiency, leaving a substantial potential for increasing performance.

As our implementation works by polling the hardware state at regular intervals, the frequency of integrity measurements will be a deciding factor for the costs to performance imposed by the system. A low polling interval will decrease the time between the occurence of an integrity compromise and its detection, but at the cost of increasing the overall workload. When deciding the polling interval, a trade-off between the immediacy of detection and the performance of the system must be made.

Immediate detection has not been set as a requirement, as an assumption has been that the system is already compromised, with the attacker having gained full administrative access to the operating system. The polling intervals used by the example policies have thus been in the range of 5 to 10 seconds, not causing any noticeable slowdowns in performance.

11.1.4 Weaknesses, vulnerabilities and attack surfaces

The history of computer security shows that attackers are quick to adapt to new technologies. It is thus imperative to look at the presented system from the angle of the attacker, trying to identify the potential weaknesses of the system.

A naive approach would be to try to hide the fact that the system is running in a virtualized environment. The motivations for hiding the use of virtualization could be both to hide the fact that the system contains an integrity checker, and to minimize the information about the environment the attacker operates in.

It has been argued that the timing properties of virtual machines makes it infeasible to effectively hide their presence [50]. In addition, by using paravirtualization, changes must be made to the guest operating system, making it extremely difficult to hide the use of virtualization. As a result, the presence of virtualization must be regarded to be known to the attacker.

The assumption that the use of virtualization makes attackers suspect the presence of additional security mechanisms is discussed and rejected in section 7.3.1.

Minimizing the attacker's knowledge of the environment in which he operates, is a reasonable goal though. While the security of the system should not be dependent on the attacker not realizing that virtualization is employed, all measures that add to the attacker's workload are of interest. However, hiding the use of virtualization would add a level of complexity to the system disproportional to the very limited amount of security gained.

Taking the virtualization environment into considerations, the attack surfaces of the system include the hypervisor, the privileged domain and the integrity analyser running inside the privileged domain.

Hypervisor

The security of the virtual environment depends on the security of the hypervisor. The hypervisor, being software, thus presents itself as a natural target for hackers to exploit.

Already in 1973, Madnick argued that the security in a virtual machine environment is *very much better* than that of a conventional operating system [52]. Madnick sees the operating systems as being too comprehensive, and thus more vulnerable to error. A similar argument is made by Goldberg in his 1974 survey of virtual machine research [39]. The increase in both complexity and size of the operating systems of today, compared to the operating systems of the mid 1970s, only strenghtens the arguments made by Madnick and Goldberg. An illustrative example is given in figure 11.1, depicting the increase of lines of code in the Windows family of operating systems from 1990 to 2002. In their book "Exploiting Software", from where the figure originates, Hoglund and McGraw observe that the number of code lines might be the only reasonable metric for predicting the number of software flaws [41].

Domain 0

The security of the Chili prototype is entirely dependent on the security of domain 0, the privileged domain in which it runs. As domain 0 runs Linux, it presents a well-known attack surface. Minimizing the exposure of domain 0 to both the other domains and the outside world is thus a key to ensuring its security. Domain 0 should provide no services except those required for running Xen, and it should not be remotely accessible. In addition, it should be stripped for unnecessary software, and its configuration should be scrutinized. Physical access should be restricted to only authorized personnel.

The above is some of the *best practices* mentioned in section 2.2.4, aimed at preventing a system from being compromised. As stated in that section, no

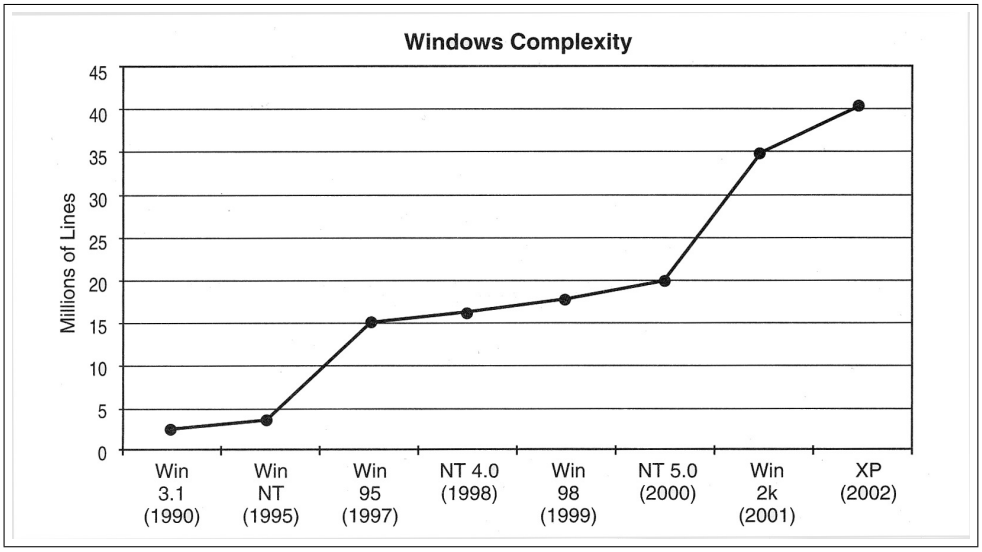

Figure 11.1: Windows complexity as measured by lines of code [41].

guarantees of the security of domain 0 can be given. However, the limited exposure of domain 0, together with the limited functionality it provides, makes the securing of the domain more feasible.

The integrity analyser

As the integrity analyser uses data from the subverted domain as input, the question must be asked of whether that input can cause the integrity analyser to fail in any way. The experiences made when testing the Chili prototype answered the question affirmatively, as following invalid pointers provided by the monitored domain caused Chili to crash (see section 10.3).

Rather than attacking the integrity analyser, a subverted domain may try evading it. As each integrity measurement applies to the state of the machine at a single point in time, the state of the machine between measurements is in fact unknown. It is thus theoretically possible that the machine enters an invalid state after a measurement, but gets restored to a valid state before the next measurement. For this to be exploited, the interval of measurements must be known to the attacker. By adding some randomness to the polling intervals, this threat may be made infeasible.

11.1.5 Generality

Although the prototype developed in this thesis uses a specific virtualization implementation, the techniques used are generic enough to work on most virtual machine environments. The implementation could be adapted to work in a full-virtualized environment, it is not dependent of the paravirtualization paradigm currently used. The x86 architecture extensions described in section 5.3.3 have greatly increased the number of open source virtulization solutions, and using one of them as a platform for an implementation of the framework presented in this thesis should not pose any fundamental issues.

11.1.6 Summarizing the virtualization platform

This discussion has shown that using virtualization as a platform for integrity analysis has real and practical benefits. By executing on the same physical piece of hardware as the monitored system, a high degree of visibility into the state of the monitored system can be achieved. The isolation provided by the virtualization environment helps in making the security of the integrity analyser plausible.

The drawbacks imposed by virtualization are first and foremost the performance overhead, but also the added complexity of the system.

11.2 Integrity analysis

The previous section argued that virtualization is beneficial both for providing isolation for the analyser, and for ensuring the completeness and correctness of the data acquired. A foundation for analysing the data is thus provided.

Assuming that the data acquired are both complete and correct, and that the integrity analyser functions correctly, the second question posed in section 1.2.1 are not fully answered: *Can the integrity of an operating system's kernel be verified?*

This section will try to provide some answers to that question, identifying some of the key issues that still remain.

11.2.1 Integrity verification

The ultimate goal of our integrity analysis is to verify the integrity of the operating system's kernel. Using familiar techniques such as signature detection and anomality detection are not sufficient. These techniques look for positive proof of compromises. When such proof is detected, it can only be deduced that the integrity of the system is broken. In other words, the integrity of the system is

falsified. The lack of such proof may indicate that the system is not compromised, but the integrity of the system has not been *verified*. For the integrity to be verified, it must be shown that the detection mechanism will detect *all* integrity breaches.

Rather than scan for signatures, the approach taken in this thesis is to monitor key parts of the kernel for modifications. By being able to attest that code and data structures have not changed, the integrity of the functionality they implement can be verified. A key assumption is that it is possible to partition the kernel into managable pieces of functionality, which can be verified independently.

11.2.2 What to check

The most naive approach to integrity checking of the kernel memory, is to make a checksum of the entire memory area allocated to the kernel. While this approach is not feasible, it provides a starting point for the discussion.

The kernel, as represented in memory, is a mix of executable code and data. While the code segments are not expected to change, large part of the data segments are.

Identifying which parts of the kernel that need to be checked in order to verify a given functionality is challenging. The system call functionality of Linux provides an illustrative example. The system call table is the key data structure for managing system calls, containing a mapping between the system calls, represented by their numbers, and the code implementing them, represented by function pointers. Assuming any changes to the system call table will be detected, the integrity of the system call table can be verified. But, on its own, this verification is of quite limited value. The only thing that can be deduced is that the content of this specific part of memory has not been changed. It still needs to be verified that this part of memory actually serves as the system call table. If malicious code has been able to redefine the memory area used for the system call table, the observation that the original area haven't been modified is of no value at all.

The above example highlights some of the difficulties of verifying the kernel's integrity. To verify a key functionality of the kernel, the verification needs not only concern itself with the individual segments of code and/or data; the entire code path needs to be verified. Special care is needed when dealing with parts of the kernel that allow for dynamic registering of function callbacks, as they are prone to hooking by malicious code.

11.2.3 Dynamic data structures

The dynamic data structures used by the kernel pose a big challenge to the integrity analysis.

Petroni et al. [58] describe how the integrity of the kernel can be compromised by manipulating dynamic data structures, without redirecting the code flow of the kernel at all. Their example deals with a common objective for kernel-level malware; the hiding of processes from the view of the system's administrators.

Typically, process hiding has been achieved through the hooking of the normal code path, either by replacing the relevant system call or, as Adore-ng (see appendix D.1) does, by replacing the function pointers used by the `proc` filesystem. Given a thorough review of the possible code paths, the relevant function pointers can be identified and monitored for changes (as is done in section 9.2.4 for system calls and in section 9.2.5 for the `proc` filesystem).

As stated above, the approach taken by Petroni et al. does not involve function pointers at all. Instead, the attack relies only on the internal representation of processes in the kernel. A process is represented by the `task_struct` data structure, also known as the *process descriptor*, containing all the information related to the process. The kernel maintains a doubly linked list of all the process descriptors, the *task-list*, with each process descriptor containing a pointer to the next and previous process descriptor. The various kernel functions needing access to all tasks can traverse the list and be sure that each process descriptor will be encountered exactly once. Linux uses a set of lists to implement a per-cpu *runqueue* for the scheduling of processes, and at any given time, each process descriptor belongs to exactly one of these *run-lists*.

The attack exploits the fact that the standard way of accessing process descriptors is through the task-list, while the scheduler uses the run-lists. The process descriptor can be removed from the task-list simply by modifying the list pointers of the previous and next process descriptors. By doing so, the process will in effect be invisible to all the kernel functions except those of the scheduler.

A variant of the technique described above is already known in kernel-level malware. The Adore-ng rootkit is implemented as a kernel module, and since the kernel maintains a list of all loaded kernel modules, Adore-ng removes itself from this list to avoid detection.

It is difficult to find a single detection technique for detecting hidden data structures. The common structure of kernel modules makes scanning memory for what appears to be hidden kernel modules a viable option. *KSTAT*, an anti-rootkit utility

for the Linux 2.4 kernel series, is known to use this technique [17]. However, as this technique uses the module structure as a basis, it could be defeated by making modifications to the structure of the hidden kernel module (see [4] for a claim of success in hiding from KSTAT).

As the attack on the process lists descibed by Petroni et al. breaks an invariant of the kernel (the process descriptors in the scheduler's run-lists should exist in the task-list), a possible counter-measure could be to regularly check if the invariant are broken. By comparing the scheduler's run-lists with the task-list of the kernel, hidden processes could be identified. However, this strategy is not without its challenges. As the lists are constantly changing, it must be ensured that no updates are in progress when inspecting the lists. Another issue would be that of performance, as affirming the kernel invariants would require lots of list traversals at frequent intervals.

11.2.4 Kernel modules

Most modern operating systems allow for the loading of additional functionality into the kernel. In section 3.3.1, the loadable kernel modules of Linux were discussed. While the run-time insertion of modules into the kernel provides for great flexibility, it also raises several security issues.

The integrity policy needs to address the fact that the functionality of the kernel is allowed to change at run-time. There are several possible approaches:

Disabling kernel modules. While this might be feasible for a special-purpose system, the use of kernel modules seems fairly entrenched in the Linux community, and the consensus seems to be that the benefits of kernel modules far outweighs any perceived risks.

Allowing kernel modules, but with run-time restrictions. Allow kernel modules, but do not allow them to change code/data that is deemed immutable. This is the approach taken in this thesis.

Allowing kernel modules, without run-time restrictions. This is the most permissive approach, giving the kernel modules free rein in extending and modifying the kernel. The safety of this approach depends entirely on the trustworthiness of

the kernel modules, necessitating mechanisms for detecting and restricting malicious kernel modules from loading. See section 4.5.1 for an example using this approach .

11.2.5 False positives and false negatives

Checksumming kernel components only reveals if the components have been modified. The integrity policies that have been made are thus rather crude, in that they see any modification as an integrity violation. Modifications that are done for legitimate reasons, causing the functionality of the system to be altered in desired ways, are thus flagged as integrity violations. This can be seen as false positives, as the changes made are in fact both known and wanted.

Likewise, the ability to manipulate data structures in malicious ways without being detected constitutes false negatives. As seen in the example above, vital kernel data can be hidden from view without using the kernel in illegal ways.

The amount of false positives will thus depend on which areas of the kernel are to be checked. By restricting the integrity checking to those parts that under no conceivable circumstances should be changed, a low rate of false positives could be ensured. However, it would increase the likelihood of false negatives, as less of the kernel would be protected against malicious modifications. Finding the right balance between the propabilities of getting false positivies or false negativies is challenging, requiring both the frequency and costs involveld to be considered.

11.2.6 Summarizing the integrity analysis

The discussion above shows that the integrity of parts of the kernel may be verified. There are however difficulties both in identifying all of the parts involved in the implementation of a given functionality, and their set of valid states. Furthermore, the code paths taken when executing that functionality may be difficult to foresee, due to the complexity and the dynamic nature of the kernel.

11.3 Future work

This section describes some possible topics for future work, including several enhancement proposals for the Chili prototype presented in this thesis.

11.3.1 Enhancing the prototype

Increase the number of policies and targets. The few policies and targets currently implemented are sufficient to illustrate most of the capabilities of the system.

In our view, constructing more targets and policies would not only broaden the protected areas of the kernel, it would most likely also provide valuable insights into the possibilities and limitations of the system.

Using multiple detection techniques. The prototype can be used to scan for malware signatures, hidden modules and other data structures of interest.

Monitor implementation. The current implementation not efficient. As of now, each policy does the monitoring by pulling the state of its targets. Targets used in multiple policies will thus be pulled once for each policy. An event-based monitoring mechanism should be implemented, delegating the monitoring of targets to the framework, allowing policies to register for notifications when the targets change. This is also discussed in section 8.2.6.

Hooking the hypervisor. By modifying the hypervisor, it could be made to notify Chili when the monitored domain attempts to write to certain memory areas, not allowing the write to take place before it has been deemed safe by Chili. As a result, Chili would not only be able to detect changes to memory areas, but also to prevent the changes from being made. It would also help efficiency, enabling notifications to be made when memory areas change, instead of the periodic pulling and checksumming of those areas.

This technique is called *interposition* by Garfinkel and Rosenblum, and is said to be implemented in their Livewire prototype implementation [37].

11.3.2 Attacks

The attack resistance of the system is hard to assess without actual trying to attack the system. Some possible venues of attack are identified in this thesis.

11.3.3 Performance

The performance overhead induced by virtualization needs to be examined through benchmarks that are entirely independent of the virtualization vendors. The additional performance penalties caused by the integrity analysis itself need also be examined.

11.3.4 Kernel synchronization

The synchronization primitives used by the kernel ensures that access to data structures is granted in a safe and controlled manner. Consider a situation where a data structure may contain several related variables, for example, a buffer and an integer containing the length of the buffer. The kernel can ensure that no thread or process may read the variables while another is in the process of updating them. The memory reads performed by our prototype are not controlled by the monitored kernel, and the possibility therefore exists that the prototype will read related memory areas as they are being updated.

11.3.5 Boot process

As part of ensuring the integrity of the virtualization environment, the bootstrapping of the system needs to be considered. A possible solution could be to make use of a trusted platform module (TPM) for verifying the bootstrapping code.

sHype is a hypervisor security architecture developed by IBM research [8]. Part of the research includes an implementation of a virtual trusted platform module (TPM) architecure for the Xen hypervisor [9].

Combining the use of a TPM with the run-time integrity monitoring described in this thesis, could prove fruitful. The two technologies are complementory, TPM focusing on load-time and this thesis' system focusing on run-time verification.

11.3.6 Placing integrity agents within the monitored host

As shown in figure 11.2, the detection and verifying mechanisms of our system are located outside the monitored system, the rationale being that the integrity of these mechanisms could not be asserted from within the monitored system.

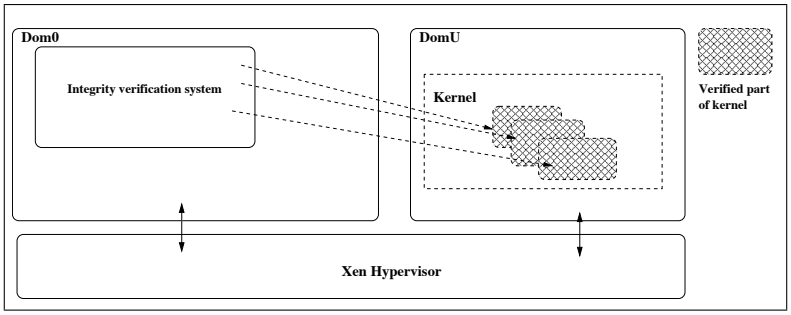

Figure 11.2: Integrity monitoring from dom0

Given that a sufficient root of trust can be established inside the kernel, the system can be augmented by placing additional mechanism inside the monitored kernel itself, as shown in figure 11.3. In other words, integrity verifying mechanisms can be placed inside the monitored host if the integrity of the functionality the mechanisms depends on can be verified. Already existing security mechanisms that are dependent on the integrity of the kernel, may again be trusted, provided that their dependencies are surveyed and verified.

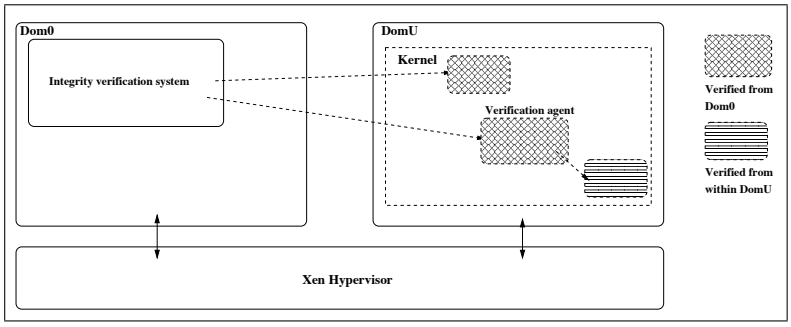

Figure 11.3: Integrity agent placed inside domU

Chapter 12

Conclusion

This chapter provides the conclusion of the thesis. First a summary of the work conducted in this thesis is given, and then the results of the work are presented.

12.1 Summary of the work conducted

The topic of this thesis has been to explore if and how the integrity of a operating system's kernel can be verified, and to show how virtualization technology can be used as a platform for the integrity analysis. As part of the research, a prototype integrity checker was also to be implemented.

The main research question from section 1.2.1 was: *How can virtualization aid in the verification of the integrity of the memory of an operating system's kernel?*

This led to these two subquestions:

1. *How can virtualization provide a platform for integrity analysis?*
2. *Can the integrity of an operating system's kernel be verified?*

To be able to answer these research questions, several diverse topics had to be given an in-depth study. First an exploration of the integrity property was conducted. Then a deeper understanding of the Linux kernel and the kernel level threats had to be acquired. In addition, the field of virtualization was studied. The outcome of these studies is presented in part I of this thesis.

Through the literature study, the key principles of a virtualization-based integrity analyser were identified, forming the basis for the design and implementation of a kernel integrity analyser prototype. This prototype provided the tool needed to try out the theoretical aspects of the research question in practice.

The results of these experiments, combined with the theoretical knowledge gained through the literature study, made a discussion of the research questions posed at the outset of the thesis possible. This discussion is presented in chapter 11.

12.2 Results

The results presented in this section are compiled both from the discussions in chapter 11 and the practical experiences with the prototype presented in chapter 10.

12.2.1 Research subquestion 1

The first of the research subquestions showed to be the easiest one to answer. The work in this thesis has shown that virtualization is indeed a capable platform for integrity checking. Both the practical experiences and the theoretical studies support this conclusion. As the discussion in section 11.1.4 points out, the Xen virtualization solution is not completely free of security challenges. However, the same section also showed that it should be possible to greatly improve many of the shortcomings.

The virtualization technology's ability to combine high visibility into the monitored kernel with strong protection against attacks from the outside, makes it a very potent foundation for an integrity checking system.

12.2.2 Research subquestion 2

The second research subquestion proved to be more complex. The key issue turned out to be the strong requirements needed for *verification* of integrity. To verify the integrity of the kernel, it is not sufficient to detect only *some* of the way a kernel can be compromised. Verification implies that the integrity system must be able to detect *all* possible integrity breaches. This simply means that a precondition for using the proposed prototype to verify a kernel's integrity, would be to identify all areas of the kernel that are not allowed to change (both code and data) with absolute certainty.

As a result, it seems unlikely that the integrity of the kernel as a whole can be verified. It should however be reasonable to assert that the integrity of many

important parts of the kernel can be verified. The approach suggested in this thesis may not be a silver bullet, but it will greatly increase the trustworthiness of the kernel.

12.3 Final comments

This thesis has showed the potential of virtualization as a platform for kernel integrity analysis. Much of that potential has been realized through the prototype developed, showing the practical implications and capabilities of the theories supporting the implementation.

It must be emphasized, though, that the integrity analyser presented in this thesis does not cover the integrity of the entire software stack of the system. Additional security mechanisms should be employed, as the presented prototype in no way should be regarded as a replacement for IDS or anti-virus software. Rather, it complements those solutions, enabling them to once again put trust in the operating system's kernel.

Since the work in this thesis has been solely based on open-source code, the authors hope that this can help provide useful insight into the implementation of a possible virtualization based integrity system, and thus making it easier for other researcher to pick up where the authors left off.

Appendix A

Loadable kernel modules

As described in chapter 3.3.1 about the Linux kernel, kernel modules are Linux's way of modifying the kernel code at run-time. In this way the kernel may be extended without the need to reboot the system.

This chapter gives a short introduction to loadable kernel modules (LKMs), and how they are made. This chapter is in no way meant to be a complete guide to LKM programming, but it is included to show the reader how relatively easy it is to change vital data structures of a running Linux kernel.

The following text only applies to the 2.6.x version of the Linux kernel, as the way kernel modules are written and compiled was altered from the 2.4.x version [69].

A.1 Making a module

Kernel modules are written in the C language. The compilation process and the libraries available to the programmer, however differ slightly from the "normal" C-programming. The usual libc-library is not available to the programmer when programming in kernel mode. The programmer must rely only on the functions available within the kernel code. Also most C-programs include a *main()* method. This is not the case in kernel modules. Instead the module requires two other methods to be present: `void module_init(void *)` and `void module_exit(void *)`. The first one is called during the installation of the module, and the second is called when the module is removed. It is however entirely up to the programmer to decide what these methods should contain.

As an example of a kernel module, a very simple kernel module written by the authors is presented below. The intention of this module is to mimic some of the behaviour of a typical rootkit (see chapter 4).

```
#include <linux/module.h> //Required for all modules
#include <linux/kernel.h> //Requires for KERN_ALERT macro
#include <linux/tty.h> //For console print
#include <linux/time.h> //For the nanosleep things
#include <linux/sched.h> //Current process
#include <asm/unistd.h> //Syscall numbers
#include <linux/syscalls.h>

#define THIS_MODULE_AUTHOR "Mads Bergdal <madsab@ifi.uio.no>, Trond A Sorby <trondasc
#define THIS_MODULE_DESC "A simple rootkit that mimics some rootkit behaviour"

//Convenient macros
#define IPRINTK(fmt, args...) \
        printk(KERN_INFO "Chili_rootkit: " fmt, ##args)
#define WPRINTK(fmt, args...) \
        printk(KERN_WARNING "Chili_rootkit: " fmt, ##args)
#define EPRINTK(fmt, args...) \
        printk(KERN_ERR "Chili_rootkit: " fmt, ##args)

//Prototypes
static unsigned long **find_sys_call_table(void);

//Globals
static unsigned long **sys_call_table;
static int (*org_nanosleep)(struct timespec *req,
                            struct timespec *);

static int hacked_nanosleep(struct timespec *req,
                            struct timespec *rem){
    //Do nothing but call org nanosleep
    return (*org_nanosleep)(req, rem);
}

static unsigned long **find_sys_call_table(void) {
  /*This is a dumb method, but it could be changed to find
    the location of the syscall table more dynamic*/

  //From "grep sys_call_table /boot/System.map"
  return (unsigned long **) 0xc02f24a0;
}

//The entry point for the module
static int __init chili_init(void){
    sys_call_table = find_sys_call_table();
    IPRINTK("Loading chili rootkit, sys call table at %p\n",
```

```
46              sys_call_table);
        IPRINTK("Hacking the sys_call_table. Nanosleep is at nbr %i\n",
                __NR_nanosleep);
        org_nanosleep = (void *)sys_call_table[__NR_nanosleep];
        IPRINTK("Org addr of nanosleep was: %p\n", org_nanosleep);
51      sys_call_table[__NR_nanosleep] = (void *)hacked_nanosleep;
        IPRINTK("New addr of nanosleep is: %p\n",
                (void *)hacked_nanosleep);

        return 0;
56  }

    static void __exit chili_cleanup(void){
        IPRINTK("Restoring sys_call_table\n");
        sys_call_table[__NR_nanosleep] = (void *)org_nanosleep;
61      IPRINTK("Addr of nanosleep is now %p\n",
                sys_call_table[__NR_nanosleep]);
    }

    module_init(chili_init);
66  module_exit(chili_cleanup);

    // Get rid of taint message by declaring code as GPL.
    MODULE_LICENSE("GPL");
    /* Who wrote this module? */
71  MODULE_AUTHOR(THIS_MODULE_AUTHOR);
    /* What does this module do */
    MODULE_DESCRIPTION(THIS_MODULE_DESC);
```

This module changes the system call table of the running kernel by swapping the pointer to the original *sys_nanosleep* with the address of the *hacked_nanosleep* function. The new function, in its current form, does nothing but call the original function, but it could have done any number of things before returning.

The *chili_init* function starts of by saving the original address of the sys_nanosleep function, and then simply inserts the address of its own function into the same position at the system call table. This is really all it takes to subvert a running kernel.

A.2 Compiling the module

Kernel modules need to be compiled a bit different than regular userspace applications. The kernel modules use the kernel's own build mechanism `kbuild` This only require the programmer to register the module as a kernel object of type *module*. This is done in the second line in the makefile below:

```
1 LINUX_DIR     = /root/src/xen-3.0.4_1-src/linux-2.6.16.33-xen
2
  obj-m += chili_rootkit.o

  all:
          $(MAKE) -C $(LINUX_DIR) M=$(PWD)  modules;
7
  clean:
          $(MAKE) -C $(LINUX_DIR) M=$(PWD)  clean
```

Note that the first line in the file above sets the location of the current linux source code. It is necessary to have access to the source code of the current running kernel to compile kernel modules.

Upon the execution of the `make` program from the same directory as this makefile, the kenrel module will be built. The resulting object file will be given a ".ko" extension.

A.3 Inserting the module

The program `insmod` is used to load modules into the kernel. It must be passed the full path of the module to be inserted. The `insmod` program does not sort out dependencies between modules. (If this is needed the more "intelligent" program `modprob` may instead be used.)

When `insmod` is executed with the kernel module name as an argument, the output in the log file should be something like:

```
xenetch:~/chilirootkit# insmod chili_rootkit.ko
Chili_rootkit: Loading chili rootkit, sys call table at c02f24a0
Chili_rootkit: IDT at ff1d9a20 Limit: 2047
Chili_rootkit: Hacking the sys_call_table. Nanosleep is at nbr 162
Chili_rootkit: Org addr of nanosleep was: c01375d0
Chili_rootkit: New addr of nanosleep is: c5008000
```

To see that the module is indeed loaded into the kernel, one may list all the currently running modules by executing the `lsmod` program:

```
debian:~/chili/trunk/resources/chilirootkit# lsmod | grep rootkit
chili_rootkit           5896  0
```

A.4 Removing the module

To remove a module currently loaded into the kernel, the program `rmmod` may be used. When used like this:

```
debian:~/chili/trunk/resources/chilirootkit# rmmod chili_rootkit
```

The output in the kernel logfile would then be something like:

```
xenetch:~/chilirootkit# rmmod chili_rootkit.ko
Chili_rootkit: Restoring sys_call_table
Chili_rootkit: Addr of nanosleep is now c01375d0
```

Appendix B

VFS objects

This chapter describes the four object types of the common file model used in the Virtual File System Switch (VFS): the superblock object, the inode object, the dentry object and the file object.

B.1 The Superblock Object

All file systems implement the superblock object, which is used to store information describing that specific file system. For disk-based file system, the superblock object usually corresponds to the `filesystem control block` stored on disk [32].

The superblock object contains an operations object, `s_op` (of type `super_operations`), which contains the methods that the kernel can invoke on a specific file system. For example, if a file system needs to allocate space for a new inode object, it would do so by invoking

```
sb->s_op->alloc_inode(inode);
```

where `sb` is a pointer to the file system's superblock, `s_op` the superblock's operations object and `alloc_inode` the desired function.

B.2 The Inode Object

The inode object represents all the information needed by the file system to handle a file. For Unix-style file systems, the inodes are represented on-disk, while for file

systems which do not have on-disk inodes, the inode objects must be constructed in whatever manner is applicable to that file system [51].

The inode object contains a number of fields describing the file's metadata. For example, the inode contains fields which describes the file type, access mode, time of the last file access and the number of blocks of the file.Each inode object has an *inode number*, which uniquely identifies the file within the file system.

The inode object also contains the `inode_operations` object, describing the functions that the VFS can invoke on an inode. This includes functions for creating, looking up and renaming inodes.

B.3 The Dentry Object

The VFS treats each directory as a file, containing a list of files and other directories. Thus, in the path `/bin/ls`, both `bin` and `ls` are files. `bin` is the special directory file and `ls` is a regular file. Since both are files, they each have an associated inode object. Files are most often reffered to with path names, and the VFS needs an efficient mechanism for mapping the components of a path to their corresponding inodes. This is achieved by using *dentry objects*, where each dentry object represents a directory entry. The path above, `/bin/ls`, consists of three dentry objects, one for the `/` root directory, one for the `bin` entry of the root directory and, finally, one for the `ls` entry of the `bin` directory.

While inodes and superblocks usually corresponds to a on-disk data structure, the dentry object only exists in memory. It is created on the fly whenever a directory entry is read into memory. The `dentry_operations` object specifies the methods associated with a dentry object.

Resolving each element in a path name into a dentry object requires considerable time, mostly because of the disk accesses required. Thus, the kernel maintains a *dentry cache*, which uses a hash table to quicly resove a given path into the associated dentry object [51].

B.4 The File Object

The file object is used to represent a file openend by a process. Because several processes can access the same file at the same time, there can be multiple file objects in existence for the same file. The `f_pos` field of the file object contains the *file pointer*, the current position in the file from which the next read or write operation will take place.

The file object does not correspond to any on-disk data; it is the *in-memory* representation of an open file, representing the process's view of the file.

The `file_operations` object contains pointers to methods performing operations such as reading and writing a file.

Appendix C

Installation of Xen

This appendix is a description of how to install the Xen hypervisor.

C.1 Base system

Firstly one have to have a linux distribution to start from. The authors have developed an affection for Debian, so we first installed a standard version of Debian Etch using a recent debian-installer (available form *http://wwww.debian.org/distrib*).

C.2 Xen hypervisor

Before the compilation of Xen starts, there is a need for some prerequisites. To install these, do this as root from the debian console:

```
apt-get install zlib1g-dev python-dev libncurses-dev bridge-utils
                iproute bzip2 libc6-xen libssl-dev libx11-dev
```

Then do:

```
wget http://bits.xensource.com/oss-xen/release \
/3.0.4-1/src.tgz/xen-3.0.4_1-src.tgz
tar zxf xen-3.0.4_1-src.tgz
```

to download and unpack the xen system.

Then start the compilation by doing:

```
cd xen-3.0.4_1-src
make world
```

This builds a standard xen kernel (pluss tools) that can be used both in Dom0 and DomU. Hopefully all goes well and one will end up with a warning saying that *latex* must be installed to build the manual. (Ignore this for now).

Then continue with:

```
make install
```

which, amongst other things, copies the kernel and the needed tools (xend, xm etc) to their "correct" directories. The next thing to do is to make an initial ramdisk:

```
update-initramfs -c -k 2.6.16.33-xen
```

C.2.1 Grub configuration

Now Grub (the bootloader) needs to be set up propperly. Edit the Grub configuration to allow the system to boot from the new kernel. Do this:

```
nano /boot/grub/menu.lst
```

and add:

```
title Xen 3.0 / XenLinux 2.6
kernel /boot/xen-3.0.gz dom0\_mem=262144
module /boot/vmlinuz-2.6.16.33-xen \
  root=/dev/hdb1 ro console=tty0
module /boot/initrd.img-2.6.16.33-xen
```

before the other kernel entries.

C.2.2 Booting Xen dom0

Now it is time to reboot the system. Hopefully one now ends up in a new system running on the xen-kernel just made. When the system comes up one can do:

```
xend start; xm list
```

and this should produce something like this:

```
Name                        ID Mem(MiB) VCPUs State  Time(s)
Domain-0                    0      256     1 r-----    34.8
```

This shows the dom0 kernel running.

C.3 User domain

To get another domain up and running, first install the debian package *xen-tools* to help install and configure new user domains. Do this:

```
apt-get install xen-tools
```

Then create a directory to contain the domU files:

```
mkdir /home/xen
```

Next, edit these fields in the file */etc/xen-tools/xen-tools.conf*:

```
dir         = /home/xen
debootstrap = 1
size        = 4Gb
memory      = 128Mb
swap        = 128Mb
fs          = ext3
dist        = etch
image       = sparse
gateway     = 192.168.2.1     # replace with your own settings
netmask     = 255.255.255.0
passwd      = 1
kernel      = /boot/vmlinuz-2.6-xen
initrd      = /boot/initrd.img-2.6.16.33-xen
mirror      = http://ftp.no.debian.org/debian/
```

Now, one can create the guest image by doing:

```
xen-create-image --hostname=xenetch --ip=192.168.2.201
```

When the image is created, it is started by doing:

```
xm create xenetch.cfg -c
```

This automatically connects to the console of the new domain. To avoid this; start without the "-c" option. To connect to the console of a domain, use the xm tool like this: `xm console <domID>`

From dom0 one can now list the domains again (`xm list`). Two domains should now be seen running.

C.4 Documentation

To install the documentation, do the following:

```
apt-get install tetex-base tetex-extra
apt-get install doxygen
apt-get install gs-common
apt-get install transfig
make dev-docs
make install-docs
```

This creates and installs the man pages of xm, xend etc.

Appendix D

Description of some rootkits

This appendix gives a brief description of the main attack techniques and capabilities of some of the kernel level rootkits mentioned in this thesis. This must not be seen as an exhaustive list of these rootkit's capabilities. Most of these rootkits are undocumented, and to get a full understanding of the inner workings of them, one has to carefully read their source code. The authors have only superficially done this for most of the mentioned rootkits.

D.1 Adore-ng

Adore-ng [1] is one of the most sophisticated kernel mode rootkits known today. It uses manipulation of the Virtual File System to return false information to the user mode tools depending on the kernel. This means that it does not alter the system call table in any way, and is therefor not detected by most kernel level malware detection tools. Adore-ng also filters all information written to e.g syslog and lastlog. It contains an advanced mechanism to hide that the network interface is in promiscuous mode. The rootkit also supports the ability to infect an already installed kernel module, making it able to get itself re-loaded as the kernel reboots. Some of the more common rootkit-features like process- and file hiding, and the ability to promote the privileges of a normal user, is also implemented. Adore-ng can be controlled both through a user mode program called *Ava*, or through a set of `echo` and `cat` commands. This rootkit has also proved to be able to run on a grate variety of linux kernels, making it highly portable.

D.2 SucKIT

The SucKIT or "Super User Control Kit" [18] is a rootkit that implements much of the same functionality as Adore-ng does. Two interesting and important differences should however be pointed out. Firstly, SucKIT is not a loadable kernel module rootkit. This rootkit inserts itself through the use of */proc/kmem* (see section 4.4.2). This means that it writes itself directly into the running kernel's memory. Second, this rootkit uses a parallell system call table (see section 4.4.2) to be able to filter information that is requested from the kernel, meaning that it, by altering the interrupt descriptor table (IDT), is able to redirect all system calls to a new version of the *system_call* function. In this way, SucKIT, as Adore-ng does, avoids all rootkit detection tools looking for changes in the original system call table.

D.3 mood-nt

mood-nt [12] is also a rootkit that installs itself via */proc/kmem*. This rootkit changes some of the system call pointers in the system call table (e.g the sys_uname system call). As most other rootkits this rootkit also implements the ability to hide files, processes, open connections etc. Although it achieves this in a less elegant manner than Adore-ng and SucKIT as it relies on making changes to the system call table.

D.4 Override

Override [13] also relies on changes to the system call table to keep itself hidden. The override rootkit was written by a computer science student for demonstration purposes only. It does not implement any surprising functionality besides the normal hide and privilege-granting mechanisms of most other rootkits. Below is a snippet from the source code of this rootkit, showing which system calls this rootkit replaces. As shown the intercepted system calls are: *getuid*, *geteuid*, *getdents64*, *chdir* and *read*.

```
int intercept_syscalls () {
#ifdef DEBUG
        printk (KERN_INFO"Intercepting Syscalls\n");
#endif
        INTERCEPT(getuid);
        INTERCEPT(geteuid);
        INTERCEPT(getdents64);
```

```
        INTERCEPT(chdir);
        INTERCEPT(read);
        // get pointers to some needed functions
        org_write = (void *) sys_call_table[__NR_write];
        org_open  = (void *) sys_call_table[__NR_open];
        org_stat  = (void *) sys_call_table[__NR_stat];
        org_brk   = (void *) sys_call_table[__NR_brk];

        return 1;
}
```

D.5 eNYeLKM

The eNYeLKM is a loadable kernel module rootkit that utilizes yet another attack technique. It does not modify the system call table nor the interrupt descriptor table, but instead modifies the system_call and sysenter_entry handlers (see section 3.3.6) to change the behavior of the kernel. It is then able to hide files, directories, processes, parts of files, connections, modules etc. It also has the ability to give a unprivileged user privileged access.

Appendix E

Chili source code

This appendix contains the relevant source code that was written during the work with this thesis. Below is a list of the files that are listet in detail throughout this appendix.

E.1 console.py

```
#! /usr/bin/env python

# This file contains the code for the user interface of the
# chili framework. It uses the cmd module to handle input
# parameters to the given commands, displaying help etc. The
# console instantiates the engine, and basically lets
# the engine do all the work.

# Author(s): Trond A Soerby, Mads Bergdal

import sys, cmd, string, socket
from chili.core.engine import Engine
from chili.core.constants import *
from chili.core import admin
from threading import Thread
import threading

class Console(cmd.Cmd, Thread):

    def __init__(self, threadFuncArray):
        if threadFuncArray == None:
            cmd.Cmd.__init__(self)
            Thread.__init__(self)
            self.engine = Engine()
```

```
            self.prompt = '> '
            version = '0.1.1'
            self.intro = 'Chili (%s), running at %s' % (version, socket.gethostname())

            #Set domain directly if domain ID is provided
            if len(sys.argv) > 1 and sys.argv[1].isdigit():
                self.do_setdom(sys.argv[1])
                self.intro = self.intro + '\nMonitoring dom ' + sys.argv[1]
                for param in sys.argv:
                    if param == '-v' or param == '--verbose':
                        self.do_setLogLevel('stdout')
                    if param == '-a' or param == '--activateAll':
                        self.do_activate('all')

        else:
            Thread.__init__(self)
            self.admin = admin.Admin()
            self.function = threadFuncArray[0]
            self.functionParams = threadFuncArray[1:]


    #Run method for a new thread
    def run(self):
        if self.function == "listDomains":
            self.admin.listDomains()
        elif self.function == "unpause":
            self.admin.unpauseDomain(self.functionParams[0])

    def do_quit(self, arg):
        self.engine.logger.info("Shutting down...")
        sys.exit(1)

    def help_quit(self):
        print "Syntax: quit"
        print "Terminates the application"

    def do_setLogLevel(self, arg):
        if arg == 'file':
            self.engine.setLogLevel(LOGONLY)
            print 'Verbose mode off'
        elif arg == 'stdout':
            self.engine.setLogLevel(LOGANDPRINT)
            print 'Verbose mode on'
        else:
            self.help_setLogLevel()

    def help_setLogLevel(self):
        print "Syntax: setLogLevel <level>"
        print "level can be 'file' which logs to file only"
```

```
74          print "or 'stdout' which logs to file and prints to stdout."

        def do_listPolicies(self, arg):
            if arg == 'active':
                returnList =  self.engine.listActivePolicies()
79              if len(returnList) == 0:
                    print "No active policies!"
                else:
                    print 'Active policies:'
                    for p in returnList:
84                      print p
            else:
                self.engine.listAvailablePolicies()

        def help_listPolicies(self):
89          print "Syntax: list [MODIFIER]"
            print "List policies (all available policies by default)"
            print " "
            print "Modifiers: "
            print "active       - list only active policies "
94
        def do_listDomains(self, arg):
            #Run this on a separate thread
            thr = Console(["listDomains", None])
            thr.start()
99          thr.join()

        def help_listDomains(self):
            print "Lists all the current running domains"

104     def do_listTargets(self, arg):
            self.engine.listTargets()

        def help_listTargets(self):
            print "Lists all the available targets"
109
        def do_setdom(self, arg):
            domID = int(arg)
            self.engine.setDomID(domID)

114     def help_setdom(self):
            print "Syntax: setdom <domID>"
            print "Sets the active Xen domain"

        def do_activate(self, arg):
119         self.engine.activatePolicy(arg)

        def help_activate(self):
            print "Syntax: activate <policy>, or activate all"
```

```
            print "Activates one policy or all policies \nif the argument 'all' is given"
124
        def do_deactivate(self, arg):
            self.engine.deactivatePolicy(arg)

        def help_deactivate(self):
129         print "Syntax: deactivate <policy> or deactivate all"
            print "Deactivates one policy or all policies \nif the argument 'all' is given"

        def do_unpause(self, arg):
            thr = Console(["unpause", arg])
134         thr.start()
            thr.join()

        def help_unpause(self):
            print "Syntax: unpause <domID>"
139         print "Unpauses a paused domain"

        # shortcuts
        do_q = do_quit

144 if __name__ == '__main__':
        con = Console(None)
        con.cmdloop(con.intro)
```

E.2 engine.py

```
#! /usr/bin/env python

# This file contains the code of the engine that makes up
# the core of the chili framework. The engine loads all
# available policies (but does not automatically activate them).
# It sets up the logging mechanism which makes it possible for
# all the policies to write their output to the log
# file. It also sets up the scheduler which are responsible for
# scheduling each of the policies method of choice in a round
    robin
# fashion. It also instantiates an inspector object through which
    it can
# supply information about targets to the requesting policies, and
    an
# administrator object through which it can send commands to the
    Xen
# hypervisor. The engine is also the one that does the actual work
    on
# requests from the console module.

# Author(s): Trond A Soerby, Mads Bergdal

import os, sys, re, time
import logging

from chili.core import inspector, oslibrary, admin
import chili.policies
from chili.taskkit import Scheduler, Task
from chili.core.constants import *

class Engine(object):

    class ChiliTask(Task.Task):
        """ChiliTasks are scheduled by the engines scheduler. For
            now it
        is only used by the policy modules.

        """
        def __init__(self, method):
            self._method = method

        def run(self):
            self._method()

    ## Init ##

    def __init__(self):
```

```
42          self._policies = {}
            self._inspector = None
            self._scheduler = None
            self.admin = None
            self.logLevel = None
47          self._setup()

        def _setup(self):
            ## logging  ##
            logging.basicConfig(level=logging.DEBUG,
52                                format='%(asctime)s %(name)-12s %(
                                    levelname)-8s %(message)s',
                                datefmt='%m-%d %H:%M:%S',
                                filename='/root/chili.log',
                                filemode='a')

57          self.logger = logging.getLogger()

            self.logger.info("Initializing application...")

            self.setLogLevel(LOGONLY)
62
            self._inspector = inspector.Inspector()

            self.admin = admin.Admin()

67          self._loadPolicies(chili.policies.__path__[0])

            self._scheduler = Scheduler.Scheduler()
            self._scheduler.start()

72          self.logger.info("Application initialized...")


        ## Administrator methods ##
        def setLogLevel(self, level):
77          """Sets the active loglevel"""
            if level == LOGONLY:
                self.logLevel = level
                self.logger.info("Log level is: log to file only")
            elif level == LOGANDPRINT:
82              self.logLevel = level
                self.logger.info("Log level is: log all and print to
                    stdout")
            else:
                print "Unknown loglevel"

87      def listDomains(self):
            """Lists the current running domains"""
```

```
        self.admin.listDomains()

    def pauseDomain(self, dom):
        """Suspends the execution of a running domain, and puts
        it in a paused state"""
        self.admin.pauseDomain(dom)

    def unpauseDomain(self, dom):
        """Unpauses a previously paused domain"""
        self.admin.unpauseDomain(dom)
        if self.logLevel == LOGANDPRINT:
            self.logger.info("Domain %i unpaused" % dom)

    ## Inspector methods ##
    def setDomID(self, domID):
        self._inspector.setDomID(domID)

    def inspect(self, target):
        return self._inspector.inspectTarget(target)


    ## Scheduler methods ##

    def schedulePeriodic(self, start, period, method, name):
        """Schedule a task to be run at <start>, and then every <
            period>
        seconds.

        """
        task = self.ChiliTask(method)
        self._scheduler.addPeriodicAction(time.time(), period,
            task, name)

    def scheduleDaily(self, hour, minute, method, name):
        """Schedule a task to be run daily at the specified time.

        """
        task = self.ChiliTask(method)
        self._scheduler.addDailyAction(hour, minute, task, name)

    def scheduleOnce(self, time, method, name):
        """Schedule a task to be run once, at the specified time.

        """
        task = self.ChiliTask(method)
        self._scheduler.addTimedAction(time, task, name)


    def unschedule(self, name):
```

```
            self._scheduler.unregisterTask(name)

        ## Policy methods ##

        def listAvailablePolicies(self):
            """Lists available policies"""
            for policy in self._policies.keys():
                print policy


        def listActivePolicies(self):
            """Lists active policies"""
            listOfActive = []
            for policy in self._policies.keys():
                if self._policies[policy]['active'] == True:
                    listOfActive.append(policy)
            return listOfActive

        def activateAllPolicies(self):
            """Activates all the available policies"""
            if self.logLevel == LOGANDPRINT:
                print 'Activating all policies'
            for policy in self._policies.keys():
                self.activatePolicy(policy)

        def activatePolicy(self, arg):
            """Activates a policy."""
            policy_sensitivity = None
            if arg == 'all':
                self.activateAllPolicies()
            else:
                args = arg.split()
                policy_name = args[0]
                if len(args) > 1:
                    policy_sensitivity = args[1]

                if self._policies.has_key(policy_name):
                    policy = self._policies[policy_name]
                    if not policy['active']:
                        policy['active'] = True
                        policy['object'].activate()
                        self.logger.info("Policy '%s' activated" %
                            policy_name)
                        if self.logLevel == LOGANDPRINT:
                            print("Policy '%s' activated" %
                                policy_name)
                            if policy_sensitivity is not None:
```

```
182                             if policy_sensitivity == HIGH or
                                    policy_sensitivity == LOW:
                                    policy['object'].setSensitivity(
                                        policy_sensitivity)
                                else:
                                    raise Exception, '%s is unknown
                                        sensitivity level' %
                                        policy_sensitivity
                    else:
187                     raise KeyError, '%s is not a valid key' %
                            policy_name

            def deactivateAllPolicies(self):
                for policy in self._policies.keys():
                    self.deactivatePolicy(policy)
192
            def deactivatePolicy(self, arg):
                """Deactivates the policy."""
                if arg == 'all':
                    self.deactivateAllPolicies()
197             else:
                    if self._policies.has_key(arg):
                        policy = self._policies[arg]
                        policy['active'] = False
                        policy['object'].deactivate()
202                     self.logger.info("Policy '%s' deactivated" % arg)
                        if self.logLevel == LOGANDPRINT:
                            print ("Policy '%s' deactivated" % arg)
                    else:
                        raise KeyError, '%s is not a valid key' %
                            policy_name
207
            def _loadPolicies(self, dir):
                """Loads policies from the given directory. Policies are
                    not
                activated upon loading, but most be explicitly activiated.

212             """

                try:
                    files = os.listdir(dir)
                except Exception, message:
217                 print message
                    sys.exit(1)

                for filename in files:
                    if filename.endswith(".py") and not filename == '
                        __init__.py' and not filename.startswith("."):
222
```

```
                module_basename = re.sub('.py$','',filename)
                module_fullname = 'chili.policies.' +
                    module_basename

                if module_fullname not in sys.modules:
                    __import__(module_fullname)
                    self.logger.info("Module %s loaded" %
                        module_fullname)

                    exec "policy = sys.modules[module_fullname].%s
                        (self)" % module_basename
                    if issubclass(policy.__class__, chili.core.
                        policy.Policy):
                        pdict = {}
                        name = module_basename
                        pdict['object'] = policy
                        pdict['active'] = False
                        self._policies[name] = pdict
                        self.logger.info("A new policy is
                            available: '%s'" % name)


    # Library methods
    def listTargets(self):
        targets = self._inspector._oslibrary.listTargets()
        print targets

    def getLibraryHelper(self, helper):
        return self._inspector._oslibrary.getHelper(helper)


## Test ##
if __name__ == '__main__':
    print "Testing engine"
    engine = Engine()
```

E.3 admin.py

```
#! /usr/bin/env python

# This module makes heavy use of the xm tool from xen. This module
# acts as a wrapper to the xm tool. This way we can utilize the xm
# tool but make our own method names and input checks.

# Author(s): Trond A Soerby, Mads Bergdal

import sys
import os

sys.path.append('/usr/lib/python')
sys.path.append('/usr/lib64/python')
from xen.xm import main

class Admin(object):
    def __init__(self):
        pass

    def listDomains(self):
        """Lists the current running domains"""
        main.main(['xm', 'list'])

    def pauseDomain(self, dom):
        """Suspends the execution of a running domain, and puts
        it in a paused state"""
        main.main(['xm', 'pause', dom])

    def unpauseDomain(self, dom):
        """Unpauses a previously paused domain"""
        main.main(['xm', 'unpause', dom])

#Test
if __name__ == '__main__':
    print "Testrun for the chili admin module!"
    admin = Admin()
    admin.listDomains()
```

E.4 oslibrary..py

```
#! /usr/bin/env python

# This file contains the code for the OS Library object. The OS
# library serves as a wrapper for all the different targets in the
# framework. It is responsible for instantiating and returning a
    fresh
# target (as it is in memory now) when requested. As the targets
    are
# highly operating system dependent, this would be the right place
     to
# implement support for operating system meta data. Currently the
    OS
# Library is only used by the inspector object.

# Author(s): Trond A Soerby, Mads Bergdal

import chili.targets
import os

class OSLibrary(object):
    def __init__(self):
        self._targets = chili.targets.__all__
        self.domID = None
        self.helpers = {}

        self._init_helpers()


    def _init_helpers(self):
        """Initialize the helpers dictionary of the OSLibrary.
        Data that may be of use for several targets may be placed
            here.
        """
        self._helper_syscalls()


    def _helper_syscalls(self):
        syscalls = []

        # syscall_table.S contains the names of all the syscalls
        filename = os.path.join(chili.targets.__path__[0], '
            syscall_table.S')

        infile = open(filename, 'r')

        # throw away
        infile.readline()
```

```
43          for line in infile:
                syscall = line.split()[1]
                syscalls.append(syscall)

            self.helpers['syscalls'] = syscalls
48

        def getHelper(self, helper):
            if self.helpers.has_key(helper):
                return self.helpers[helper]
53          else:
                raise ValueError, '%s is not in the helpers dictionary
                    ' % helper


        def setDomID(self, domID):
58          self.domID = domID

        def getTarget(self, targetType):
            """Returns a target object of the given type"""

63          if self.domID is None:
                raise ValueError, 'DomID is not set'
            else:
                if targetType in self._targets:
                    __import__('chili.targets.' + targetType)
68
                    exec "target = chili.targets.%s.%s('a_name', self.
                        domID)" % (targetType, targetType)
                    return target
                else:
                    raise ValueError, '%s is not a valid target' %
                        targetType
73
        def listTargets(self):
            """Returns a list containing the available target types"""
            return self._targets
```

E.5 inspector.py

```
#! /usr/bin/env python

# The inspector object is responsible for returning  a current
# representation from memory of a given target. It uses the
# OS Library to instantiate a new target object of the given type.

# Author(s): Trond A Soerby, Mads Bergdal

import oslibrary

class Inspector(object):
    def __init__(self):
        self._oslibrary = oslibrary.OSLibrary()

    def setDomID(self, domID):
        self._oslibrary.setDomID(domID)

    def inspectTarget(self, target):
        return self._oslibrary.getTarget(target)
```

E.6 policy.py

```
#! /usr/bin/env python

# This file is the interface for the policies  we are using in
# the chili framework.

# Author(s): Trond A Soerby, Mads Bergdal

from chili.core.constants import *

class Policy(object):

    def __init__(self, engine):
        self.engine = engine
        self.setup()
        #Default to HIGH for all
        self.sensitivity = HIGH

    def setup(self):
        print "I'm the policy base class..."

    def activate(self):
        raise NotImplementedError, "method not implemented by
            subclass!"

    def deactivae(self):
        raise NotImplementedError, "method not implemented by
            subclass!"

    def doMyStuff(self):
        raise NotImplementedError, "method not implemented by
            subclass!"

    def setSensitivity(self, sensitivity):
        raise NotImplementedError, "method not implemented by
            subclass!"


    def log(self, msg):
        if self.engine.logLevel == LOGANDPRINT:
            self._log_and_print(msg)
        else:
            self._log_only(msg)


    def _log_only(self, msg):
        self.engine.logger.info(msg)
```

```
    def _log_and_print(self, msg):
45      self.engine.logger.info(msg)
        print(msg)
```

E.7 IDTMonitor.py

```
#! /usr/bin/env python

# This file contains the code for the Interupt Descriptor Table (IDT)
# policy monitor. It uses the IDT target to get information about the current
# IDT of the domU. When the checksum of the  table
# is changed the policy signals the engine to pause the user domain.

#   Author(s): Trond A Soerby, Mads Bergdal

import time
import chili.core.policy
from chili.core.constants import *

class IDTMonitor(chili.core.policy.Policy):

    def setup(self):
        # Sets the seconds between each poll
        self.pollingInterval = 5

    def setSensitivity(self, sensitivity):
        self.sensitivity = sensitivity

    def activate(self):
        self.sensitivity = LOW
        self.target_old = self.engine.inspect("IDT")
        self.engine.schedulePeriodic(time.time(),
                                     self.pollingInterval,
                                     self.doMyStuff,
                                     'IDTMonitor_doMyStuff')

    def deactivate(self):
        self.engine.unschedule('IDTMonitor_doMyStuff')

    def decideWhatToDo(self):
        self.log(' ')
        self.log('IDTMonitor: IDT table has changed!')
        self.log('IDTMonitor: MD5 was: %s' % self.digest_old)
        self.log('IDTMonitor: MD5 is : %s' % self.digest_new)

        # Check log level if the domain should be paused
        if self.sensitivity == HIGH:
            self.log('IDTMonitor: Domain %s was paused according
                to policy' %
                                      self.target_old.domID)
```

```
            self.engine.pauseDomain(self.target_old.domID)

    def doMyStuff(self):
        self.target_new = self.engine.inspect("IDT")
48
        self.state_new = self.target_new.getState()
        self.state_old = self.target_old.getState()

        self.digest_new = self.state_new.getDigest()
53      self.digest_old = self.state_old.getDigest()

        if not self.digest_new == self.digest_old:
            # The IDT has changed
            self.decideWhatToDo()
```

E.8 SyscallMonitor.py

```
#! /usr/bin/env python

# This file contains the code for the Syscalls policy monitor. It uses
# the Syscalls target to get information about the current Syscall
# procedures of the domU. When one of the addresses in the syscall table
# is changed, the digest of the array of Syscall structs will change. If
# the digest of one of the service routines pointed to by these
# addresses changes the policy will notice this too. Depending on the
# loglevel defined by the engine, the policy will signals the engine to
# pause the user domain. Information about which system call structure
# was changed is written to the log.

#   Author(s): Trond A Soerby, Mads Bergdal

import time
import chili.core.policy
import sys
from chili.core.constants import *

class SyscallMonitor(chili.core.policy.Policy):

    def setup(self):
        pass

    def activate(self):
        self.sensitivity = LOW

        # System call table targets
        self.syscalltable_org = self.engine.inspect("SysCallTable")
        self.syscalltable_cur = self.syscalltable_org

        self.syscalltable_size = self.syscalltable_org.getState().size
        self.syscalltable_table = self.syscalltable_org.getState().table

        # Service routines targets
        self.serviceroutinearray_org = self.engine.inspect("ServiceRoutineArray")
```

```
37          self.serviceroutinearray_org.setSCT(self.syscalltable_org.
                getState().table, self.syscalltable_size)
            self.serviceroutinearray_cur = self.
                serviceroutinearray_org

            self.serviceroutine_names = self.engine.getLibraryHelper('
                syscalls')

42          self.engine.schedulePeriodic(time.time(),
                                         5,
                                         self.doMyStuff,
                                         'SyscallMonitor_doMyStuff')


47
        def deactivate(self):
            self.engine.unschedule('SyscallMonitor_doMyStuff')


52      ### Helper functions ###
        def logDetailsSCT(self):
            state_org = self.syscalltable_org.getState()
            state_cur = self.syscalltable_cur.getState()

57          for i in range(0, state_cur.size):
                sc_org = state_org.getAt(i)
                sc_cur = state_cur.getAt(i)
                if sc_org != sc_cur:
                    self.log('SyscallMonitor: The address of syscall %
                        s changed' % self.serviceroutine_names[i])
62                  self.log('          Original address: %x' %
                        sc_org)
                    self.log('          Current address : %x' %
                        sc_cur)


        def logDetailsSRA(self):
67          for i in range(0, self.syscalltable_size):
                sr_org = self.serviceroutinearray_org.getState().getAt
                    (i).getDigest()
                sr_cur = self.serviceroutinearray_cur.getState().getAt
                    (i).getDigest()

                if sr_org != sr_cur:
72                  self.log(' ')
                    self.log('SyscallMonitor: The hash of the service
                        function for %s changed' % self.
                        serviceroutine_names[i])
                    self.log('          Original hash: %x' % sr_org)
                    self.log('          Current  hash: %x' % sr_cur)
```

```
77
        def decideWhatToDo(self, error, output=None):
            if error == 'SCT_CHANGED':
                self.log(' ')
82              self.log('SyscallMonitor: The system call table has
                    changed.')
                self.logDetailsSCT()

            if error == 'SR_ARRAY_CHANGED':
                self.log(' ')
87              self.log('SyscallMonitor: A service routine has been
                    modified.')
                self.log(msg)
                self.logDetailsSRA()


92          # Check if the domain should be paused
            if self.sensitivity == HIGH:
                msg = 'SyscallMonitor: Domain %s was paused according
                    to policy' % self.syscalltable_cur.domID
                self.log(msg)
                self.engine.pauseDomain(self.syscalltable_cur.domID)
97

        def doMyStuff(self):
            # Gets a new system call table
            self.syscalltable_cur = self.engine.inspect("SysCallTable"
                )
102
            if self.syscalltable_cur.getState().digest != self.
                syscalltable_org.getState().digest:
                self.decideWhatToDo('SCT_CHANGED')
            else:
                self.serviceroutinearray_cur = self.engine.inspect("
                    ServiceRoutineArray")
107             self.serviceroutinearray_cur.setSCT(self.
                    syscalltable_org.getState().table, self.
                    syscalltable_size)
                digest_cur = self.serviceroutinearray_cur.getState().
                    getDigest()
                digest_org = self.serviceroutinearray_org.getState().
                    getDigest()

                if digest_cur != digest_org:
112                 self.decideWhatToDo('SR_ARRAY_CHANGED')
```

E.9 ProcMonitor.py

```
#! /usr/bin/env python

# This file contains the code for the Proc file system policy
    monitor.
# It uses the Proc target to get information about the current
# Proc file system of the domU. When the checksum of the part of
    proc
# that is supposed to be static  is changed the policy signals the
# engine to pause the user domain.

# Author(s): Trond A Soerby, Mads Bergdal

import time
import chili.core.policy
from chili.core.constants import *

class ProcMonitor(chili.core.policy.Policy):

    def setup(self):
        #Sets the secounds between poll
        self.pollingInterval = 5

    def setSensitivity(self, sensitivity):
        self.sensitivity = sensitivity

    def activate(self):
        self.sensitivity = LOW
        self.proc_base = self.engine.inspect("Proc")
        self.state_base = self.proc_base.getState()
        self.engine.schedulePeriodic(time.time(),
                                     self.pollingInterval,
                                     self.doMyStuff,
                                     'ProcMonitor_doMyStuff')

    def deactivate(self):
        self.engine.unschedule('ProcMonitor_doMyStuff')

    def doMyStuff(self):
        printAlso = 0
        if self.engine.logLevel == LOGANDPRINT:
            printAlso = 1

        self.proc_current = self.engine.inspect("Proc")
        state_current = self.proc_current.getState()

        if not state_current.digest == self.state_base.digest:
```

```
            self.engine.logger.info('ProcMonitor: Proc has changed
                !')
46          self.engine.logger.info('ProcMonitor: MD5 of base proc
                : %s' % self.state_base.digest)
            self.engine.logger.info('ProcMonitor: MD5 of current
                proc : %s' % state_current.digest)
            if printAlso:
                print('ProcMonitor: Proc has changed!')
                print('ProcMonitor: MD5 of base proc: %s' % self.
                    state_base.digest)
51              print('ProcMonitor: MD5 of current proc : %s' %
                    state_current.digest)

            #Check log level if the domain should be paused
            if self.sensitivity == HIGH:
                self.engine.logger.info('ProcMonitor: Domain %s
                    was paused according to policy' % self.
                    proc_base.domID)
56              if printAlso:
                    print('ProcMonitor: Domain %s was paused
                        according to policy' % self.proc_base.
                        domID)
                self.engine.pauseDomain(self.proc_base.domID)
```

E.10 KernelVsyscall.py

```
#! /usr/bin/env python

# Author(s): Trond A Soerby, Mads Bergdal

import time
import chili.core.policy
from chili.core.constants import *

class KernelVsyscall(chili.core.policy.Policy):

    def setup(self):
        #Sets the secounds between each poll
        self.pollingInterval = 10

    def setSensitivity(self, sensitivity):
        self.sensitivity = sensitivity

    def activate(self):
        self.sensitivity = LOW

        self.kernel_vsyscall_org = self.engine.inspect("MemoryArea
            ")
        self.kernel_vsyscall_org.setMemoryArea(0, '
            __kernel_vsyscall', 10)


        self.engine.schedulePeriodic(time.time(),
                                     self.pollingInterval,
                                     self.doMyStuff,
                                     'KernelVsyscall_doMyStuff')

    def deactivate(self):
        self.engine.unschedule('KernelVsyscall_doMyStuff')

    def doMyStuff(self):
        self.kernel_vsyscall_cur = self.engine.inspect("MemoryArea
            ")
        self.kernel_vsyscall_cur.setMemoryArea(0, '
            __kernel_vsyscall', 10)
        state_cur = self.kernel_vsyscall_cur.getState()
        state_org = self.kernel_vsyscall_org.getState()

        if not state_cur.digest == state_org.digest:
            self.log('KernelVsyscall: The __kernel_vsyscall
                function has changed!')
            self.log('KernelVsyscall: MD5 of function was   : %s'
                % state_org.digest)
```

```
42              self.log('KernelVsyscall: MD5 of function is now: %s'
                    % state_cur.digest)

                #Check log level if the domain should be paused
                if self.sensitivity == HIGH:
                    self.log('KernelVsyscall: Domain %s was paused
                        according to policy' %
47                                          self.kernel_vsyscall_org.
                                                domID)
                    self.engine.pauseDomain(self.kernel_vsyscall_org.
                        domID)
```

E.11 target.py

```
#!/usr/bin/env/ python

# This file is the interface for the targets we are using in the
    chili framework.

# Author(s): Trond A Soerby, Mads Bergdal

class Target(object):

    def __init__(self, name, domID):
            self._name = name
            self.domID = domID
            self.setup()

    def setup(self, xai):
       raise NotImplementedError, "Setup method not implemented by
            subclass!"

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""
        raise NotImplementedError, "show_state method not
            implemented by subclass!"
```

E.12 SysCallTable.py

```
#!/usr/bin/env/ python
"""Keeps track of processes"""

from chili.core.target import Target
import sys, os, ctypes
import sct_target_data


class SysCallTable(Target):

    def setup(self):
        self.sct_data = sct_target_data.SysCallTable()
        sct_target_data.get_state(self.domID, self.sct_data)

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""

        return self.sct_data


    def update(self):
        """Responsible for updating the inner state of
            this target from the memory"""
        raise NotImplementedError, "update method not implemented
            by subclass!"
```

E.13 sct_target_data.c

```
1 /*This file contains the c implementation of the syscalls target.
      It
2 uses a struct that contians an array of Syscall structs, and a MD5
  digest of this array. The Syscall structs contain the original
      address of
  the procedure implementing this syscall and the digest of this
      procedure
  as it is in memory now. This file uses the Xenaccess library
      functions
  to get information from domU.*/
7
  /* Author(s): Trond A Soerby, Mads Bergdal */

  #include <stdlib.h>
  #include <string.h>
12 #include <errno.h>
  #include <sys/mman.h>
  #include <asm/unistd.h>
  #include <stdio.h>
  #include <xenaccess.h>
17 #include <xa_private.h>
  #include <libca.h>

  #include <sct_target_data.h>


22
  void make_digest(struct SysCallTable* s, ca_mmap_t* m) {
      md5_state_t state;
      md5_byte_t digest[16];
      int di;
27    char *hex_output;
      char out[16*2 + 1];


      md5_init(&state);
32
      md5_append(&state, (const md5_byte_t *) m->start_addr,
                 NR_syscalls*sizeof(s->table[0]));

      md5_finish(&state, digest);
37
      for (di = 0; di < 16; ++di) {
          sprintf(hex_output + di * 2, "%02x", digest[di]);
      }
  }
42
```

```
void map_sct(xa_instance_t* instance, ca_mmap_t* m,
             struct SysCallTable* s) {

47      char sct_symbol[] = "sys_call_table";
        uint32_t table_start;
        size_t entry_size = sizeof(s->table[0]);
        int i;

52

        /* get the vaddr of the start of the sys_call_table */
        linux_system_map_symbol_to_address(instance, sct_symbol,
                                           &table_start);
57
        ca_map_address_range(instance, table_start,
                             table_start + NR_syscalls*entry_size, m);

        /* copy the table entries */
62      for(i=0; i<NR_syscalls; i++){
            memcpy(&s->table[i], m->start_addr+i*entry_size,
                entry_size);
        }
    }

67  void get_state(int dom, struct SysCallTable* sct){
        xa_instance_t xai;
        ca_mmap_t map;


72      /* initialize the xen access library */
        if (xa_init(dom, &xai) == XA_FAILURE){
            perror("failed to init XenAccess library");
            goto error_exit;
        }
77
        map_sct(&xai, &map, sct);
        make_digest(sct, &map);
        sct->size = NR_syscalls;

82      ca_unmap(&map);

    error_exit:
        /* cleanup any memory associated with the XenAccess instance
            */
        xa_destroy(&xai);
87


    }
```

E.14 sct_target_data.h

```
#include <targets_data.h>


struct SysCallTable {
    int size;
    uint32_t table[NR_syscalls];
    char digest[16*2 + 1];
};


void get_state(int dom, struct SysCallTable *);
```

E.15 sct_target_data.i

```
/*This is the file used by the swig system do turn c-code
into python. This file tells swig what we want to wrap.
Here we tell swig to wrap all the things in the header
file.*/

%module sct_target_data
%{
#include <sct_target_data.h>
%}


%extend SysCallTable{
   uint32_t getAt(int index){
     return self->table[index];
   }
}

%include "../../targets_data.h"
%include "../sct_target_data.h"
```

E.16 ServiceRoutineArray.py

```
#!/usr/bin/env/ python
"""Keeps track of processes"""

from chili.core.target import Target
import sys, os, ctypes
import sr_target_data


class ServiceRoutineArray(Target):


    def setup(self):
        self.sr_data = sr_target_data.ServiceRoutineArray()



    def setSCT(self, table, len):
        self._table_set = True
        sr_target_data.get_state(self.domID, self.sr_data, table,
            len)


    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""

        if self._table_set:
            return self.sr_data
        else:
            return None



    def update(self):
        """Responsible for updating the inner state of
            this target from the memory"""
        raise NotImplementedError, "update method not implemented
            by subclass!"
```

E.17 sr_target_data.c

```
1 /*This file contains the c implementation of the syscalls target.
      It
2 uses a struct that contians an array of Syscall structs, and a MD5
  digest of this array. The Syscall structs contain the original
      address of
  the procedure implementing this syscall and the digest of this
      procedure
  as it is in memory now. This file uses the Xenaccess library
      functions
  to get information from domU. */
7
  /* Author(s): Trond A Soerby, Mads Bergdal */

  #include <stdlib.h>
  #include <string.h>
12 #include <errno.h>
  #include <sys/mman.h>
  #include <asm/unistd.h>
  #include <stdio.h>
  #include <xenaccess.h>
17 #include <xa_private.h>
  #include <libca.h>

  #include <sr_target_data.h>

22
  void hash_sr(xa_instance_t* instance, struct ServiceRoutine* sr)
      {
      md5_state_t state;
      ca_mmap_t map;

27
      ca_map_address_range(instance, sr->addr,
                           sr->addr + sr->length, &map);


32    md5_init(&state);
      md5_append(&state, (const md5_byte_t *) map.start_addr, sr->
          length);
      md5_finish(&state, sr->digest);

      ca_unmap(&map);
37 }

  void hash_sra(struct ServiceRoutineArray* sra, int sct_size) {
      md5_state_t state;
```

```
42      md5_init(&state);
        md5_append(&state, (const md5_byte_t *) sra->routines,
                   sct_size*sizeof(struct ServiceRoutine));
        md5_finish(&state, sra->digest);
    }
47


    void get_state(int dom, struct ServiceRoutineArray* sra, uint32_t
        sct[], int sct_size){
        xa_instance_t xai;
52      int i;

        /* initialize the xen access library */
        if (xa_init(dom, &xai) == XA_FAILURE){
            perror("failed to init XenAccess library");
57          goto error_exit;
        }

        struct ServiceRoutine *sr;

62      for(i=0; i<sct_size; i++) {
            sr = &sra->routines[i];
            sr->addr = sct[i];
            sr->length = SR_LENGTH;
            hash_sr(&xai, sr);
67

        }

        hash_sra(sra, sct_size);

72  error_exit:
        /* cleanup any memory associated with the XenAccess instance
            */
        xa_destroy(&xai);


77  }
```

E.18 sr_target_data.h

```
#include <targets_data.h>

#define SR_LENGTH 31

struct ServiceRoutine{
    uint32_t addr;
    int length;
    md5_byte_t digest[16];
};

struct ServiceRoutineArray{
    struct ServiceRoutine routines[NR_syscalls];
    md5_byte_t digest[16];
};


void get_state(int dom, struct ServiceRoutineArray *, uint32_t[],
    int);
```

E.19 sr_target_data.i

```
/*This is the file used by the swig system do turn c-code
into python. This file tells swig what we want to wrap.
Here we tell swig to wrap all the things in the header
file.*/

%module sr_target_data
%{
#include <sr_target_data.h>
%}

%extend ServiceRoutine{
  char * getDigest(){
    static char hex_output[16*2 + 1];
    int i = 0;
    for (i = 0; i < 16; ++i) {
       sprintf(hex_output + i * 2, "%02x", self->digest[i]);
    }
    return hex_output;
  }
}

%extend ServiceRoutineArray{
  char * getDigest(){
    static char hex_output[16*2 + 1];
    int i = 0;
    for (i = 0; i < 16; ++i) {
       sprintf(hex_output + i * 2, "%02x", self->digest[i]);
    }
    return hex_output;
  }
}

%extend ServiceRoutineArray{
  struct ServiceRoutine getAt(int index){
    return self->routines[index];
  }
}



%include "../../targets_data.h"
%include "../sr_target_data.h"
```

E.20 IDT.py

```
#!/usr/bin/env/ python

# This file contains the code for the Interrupt descriptor table (IDT)
# target. It imports the python module idt_target_data, which is the python
# wrapper for the C-implementation of this target. The self.table is an
# object which represents a struct i c. This struct contains an array of
# trap_info_t structs (Which contains info about each trap vector) and a digest
# of this array.

# Author(s): Trond A Soerby, Mads Bergdal

from chili.core.target import Target
import sys, os, ctypes
import idt_target_data

class IDT(Target):
    def setup(self):
        self.table = idt_target_data.IDT_table()
        idt_target_data.getState(self.domID, self.table)

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""
        return self.table

    def update(self):
        """Responsible for updating the inner state of
        this target from the memory"""
        raise NotImplementedError, "update method not implemented by subclass!"
```

E.21 idt_target_data.c

```
/*This file contains the c implementation of the interrupt
descriptor table (IDT) target. It uses a struct that contains
an array of trap_info_t structs (Which contains info about
each trap vector) and a digest of this array as it is in memory
now. This file uses the Xenaccess library functions to get
information from domU.*/

/* Author(s): Trond A Soerby, Mads Bergdal */

#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdio.h>
#include "md5.h"
#include <xenaccess/xenaccess.h>
#include "libca.h"
#include <idt_target_data.h>

void hash_idt_table(uint32_t table_start,
                    IDT_table* return_struct,
                    xa_instance_t* xai){

     md5_state_t state;
    uint32_t end_addr;
    trap_info_t tmp_trap_info;
    int i= 0;

    end_addr = table_start+(NR_IDT_ENTRIES*sizeof(trap_info_t));

    //Copy the idt entries
    for(i=0; i<NR_IDT_ENTRIES; i++){
        memcpy(&tmp_trap_info,
               (void *)table_start+i*sizeof(trap_info_t), sizeof(
                   trap_info_t));
        return_struct->idt_entries[i] = tmp_trap_info;
    }

    md5_init(&state);
    md5_append(&state, (const md5_byte_t *) ((void *)table_start),
               NR_IDT_ENTRIES*sizeof(trap_info_t));
    md5_finish(&state, return_struct->digest);
}

void getState(int dom, IDT_table *idt_table) {
    xa_instance_t xai;
    vcpu_guest_context_t ctxt;
```

```
    /* initialize the xen access library */
    if (xa_init(dom, &xai) == XA_FAILURE){
        perror("failed to init XenAccess library");
        goto error_exit;
    }

    //Get the address of the vcpu_context which contains the idt
        table
    if (xc_vcpu_getcontext(xai.xc_handle, xai.domain_id, 0, &ctxt)
        < 0) {
        perror("failed to get vcpu context");
        goto error_exit;
    };

    hash_idt_table((uint32_t)&ctxt.trap_ctxt, idt_table, &xai);

error_exit:
    /* cleanup any memory associated with the XenAccess instance
        */
    xa_destroy(&xai);
}
```

E.22 idt_target_data.h

```
#include <xenaccess/xa_private.h>
#include <targets_data.h>

#define SYSCALL_VECTOR 128
#define NR_IDT_ENTRIES 256


//Provided as a reference. Originally defined in xen.
//struct trap_info {
//      uint8_t       vector;  /* exception vector */
//      uint8_t       flags;   /* 0-3: privilege level */
//      uint16_t      cs;      /* code selector */
//      unsigned long address; /* code offset  */
//  };
//typedef struct trap_info trap_info_t;

typedef struct IDT_table{
    md5_byte_t digest[16]; /*The hash of all IDT entries*/
    trap_info_t idt_entries[NR_IDT_ENTRIES];
}IDT_table;

void getState(int dom, IDT_table *table);
```

E.23 idt_target_data.i

```
/*This is the file used by the swig system do turn c-code
into python. This file tells swig what we want to wrap.
Here we tell swig to wrap all the things in the header
files as well as the functions defined in this file.*/

%module idt_target_data
%{
#include <idt_target_data.h>
%}

%extend IDT_table{
  uint32_t getAddressAt(int index){
    return self->idt_entries[index].address;
  }

  char * getDigest(){
    static char hex_output[16*2 + 1];
    int i = 0;
    for (i = 0; i < 16; ++i) {
       sprintf(hex_output + i * 2, "%02x", self->digest[i]);
    }
    return hex_output;
  }
}

%include "../../targets_data.h"
%include "../idt_target_data.h"
```

E.24 Proc.py

```
#!/usr/bin/env/ python

# This file contains the code for the Proc filesystem target.
# It imports the python module proc_target_data, which is the
    python
# wrapper for the C-implementation of this target. The self.
    proc_data
# is an object which represents a struct i c. This struct
    contatins three
# other structs: proc_dir_entry, inode_operations and
    file_operations
# (which all represents internal structures of the Proc file
    system)
# and a digest of these structures as they are in current memory.

# Author(s): Trond A Soerby, Mads Bergdal

from chili.core.target import Target
import sys, os, ctypes
import proc_target_data

class Proc(Target):

    def setup(self):
        self.proc_data = proc_target_data.Proc()
        proc_target_data.get_state(self.domID, self.proc_data)

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""

        return self.proc_data


    def update(self):
        """Responsible for updating the inner state of
            this target from the memory"""
        raise NotImplementedError, "update method not implemented
            by subclass!"
```

E.25 proc_target_data.c

```
/*This file contains the c implementation of the proc filesystem
target. It uses a struct that contatins three other structs:
proc_dir_entry, inode_operations and file_operations (which all
represents internal structures of the Proc file system) and a
    digest
of these structures as they are in current memory. This file
    contains
methods to fill these structures and to make a digest of them.
    This
file uses the Xenaccess library functions to get information from
domU. */

/* Author(s): Trond A Soerby, Mads Bergdal */

#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdio.h>
#include <xenaccess/xenaccess.h>
#include <xenaccess/xa_private.h>
#include <libca.h>
#include <proc_target_data.h>

void make_digest(struct Proc *p) {
    md5_state_t state;
    md5_byte_t digest[16];
    int di;

    char *hex_output = p->digest;

    md5_init(&state);

    md5_append(&state, (const md5_byte_t *) &p->pde.proc_iops,
            sizeof(struct inode_operations *));
    md5_append(&state, (const md5_byte_t *) &p->pde.proc_fops,
            sizeof(struct file_operations *));
    md5_append(&state, (const md5_byte_t *) &p->iops,
            sizeof(struct inode_operations));
    md5_append(&state, (const md5_byte_t *) &p->fops,
            sizeof(struct file_operations));

    md5_finish(&state, digest);

    for (di = 0; di < 16; ++di) {
        sprintf(hex_output + di * 2, "%02x", digest[di]);
    }
```

```
}

void fill_pde(xa_instance_t *instance, struct Proc *p) {
    ca_mmap_t map;
    uint32_t start, end;

    /* get the vaddr of proc_root */
    linux_system_map_symbol_to_address(instance, "proc_root", &
        start);
    end = start + sizeof(struct proc_dir_entry);

    ca_map_address_range(instance, start, end, &map);
    memcpy(&p->pde, map.start_addr, sizeof(struct proc_dir_entry))
        ;

    ca_unmap(&map);

}

void fill_iops(xa_instance_t* instance, struct Proc *p) {
    char *memory;
    uint32_t offset;

    memory = xa_access_virtual_address(instance,
                                       (uint32_t) p->pde.proc_iops
                                           ,
                                       &offset);

    memcpy(&p->iops, memory + offset, sizeof(struct
        inode_operations));

    /* sanity check to unmap shared pages */
    if (memory) munmap(memory, XA_PAGE_SIZE);
}

void fill_fops(xa_instance_t* instance, struct Proc *p) {
    char *memory;
    uint32_t offset;

    memory = xa_access_virtual_address(instance,
                                       (uint32_t) p->pde.proc_fops
                                           ,
                                       &offset);

    memcpy(&p->fops, memory + offset, sizeof(struct
        file_operations));

    /* sanity check to unmap shared pages */
    if (memory) munmap(memory, XA_PAGE_SIZE);
```

```
}

void get_state (int dom, struct Proc *proc) {
    xa_instance_t xai;

    /* initialize the xen access library */
    if (xa_init(dom, &xai) == XA_FAILURE){
        perror("failed to init XenAccess library");
        goto error_exit;
    }

    /* put the proc_dir_entry into proc */
    fill_pde(&xai, proc);

    fill_iops(&xai, proc);
    fill_fops(&xai, proc);


#ifdef DEBUG
    printf("proc_iops: 0x%.8x\n", (uint32_t) proc->pde.proc_iops);
    printf("proc_fops: 0x%.8x\n", (uint32_t) proc->pde.proc_fops);
    print_hex((unsigned char *) &proc->pde, sizeof(proc->pde));

    printf("proc_iops_lookup: 0x%.8x\n", (uint32_t) proc->iops.
        lookup);
    print_hex((unsigned char *) &proc->iops, sizeof(proc->iops));
    print_hex((unsigned char *) &proc->fops, sizeof(proc->fops));
#endif

    /* calculate the md5 hash for the immutable fields */
    make_digest(proc);


    /* #define DEBUG */
#ifdef DEBUG
    printf("%s\n", proc->digest);
#endif

error_exit:
    /* cleanup any memory associated with the XenAccess instance
        */
    xa_destroy(&xai);
}
```

E.26 proc_target_data.h

```
1 #include <targets_data.h>
2 #include <linux/types.h>

  typedef struct { volatile int counter; } atomic_t;

  struct proc_dir_entry {
7         unsigned int low_ino;
          unsigned short namelen;
          const char *name;
      /*  mode_t mode; */
      /*  nlink_t nlink; */
12        __kernel_mode_t mode;
          __kernel_nlink_t nlink;
          uid_t uid;
          gid_t gid;
          unsigned long size;
17        struct inode_operations * proc_iops;
          struct file_operations * proc_fops;
      /*  get_info_t *get_info; */
          void *get_info;
          struct module *owner;
22        struct proc_dir_entry *next, *parent, *subdir;
          void *data;
      /*  read_proc_t *read_proc; */
      /*  write_proc_t *write_proc; */
          void *read_proc;
27        void *write_proc;
          atomic_t count;         /* use count */
          int deleted;            /* delete flag */
          void *set;
  };
32
  struct inode_operations {
          int (*create) (void *,void *,int, void *);
          void * (*lookup) (void *,void *, void *);
          int (*link) (void *,void *,void *);
37        int (*unlink) (void *,void *);
          int (*symlink) (void *,void *,const char *);
          int (*mkdir) (void *,void *,int);
          int (*rmdir) (void *,void *);
          int (*mknod) (void *,void *,int,dev_t);
42        int (*rename) (void *, void *,
                         void *, void *);
          int (*readlink) (void *, char *,int);
          void * (*follow_link) (void *, void *);
          void (*put_link) (void *, void *, void *);
47        void (*truncate) (void *);
```

```
        int (*permission) (void *, int, void *);
        int (*setattr) (void *, void *);
        int (*getattr) (void *, void *, void *);
        int (*setxattr) (void *, const char *, void *,size_t,int);
52      ssize_t (*getxattr) (void *, const char *, void *, size_t)
            ;
        ssize_t (*listxattr) (void *, char *, size_t);
        int (*removexattr) (void *, const char *);
        void (*truncate_range)(void *, loff_t, loff_t);
};
57
struct file_operations {
        void *owner;
        loff_t (*llseek) (void *, loff_t, int);
        ssize_t (*read) (void *, void *, size_t, loff_t *);
62      ssize_t (*aio_read) (void *, void *, size_t, loff_t);
        ssize_t (*write) (void *, void *, size_t, loff_t *);
        ssize_t (*aio_write) (void *, void *, size_t, loff_t);
        int (*readdir) (void *, void *, int);
        unsigned int (*poll) (void *, void *);
67      int (*ioctl) (void *, void *, unsigned int, unsigned long)
            ;
        long (*unlocked_ioctl) (void *, unsigned int, unsigned
            long);
        long (*compat_ioctl) (void *, unsigned int, unsigned long)
            ;
        int (*mmap) (void *, void *);
        int (*open) (void *, void *);
72      int (*flush) (void *);
        int (*release) (void *, void *);
        int (*fsync) (void *, void *, int datasync);
        int (*aio_fsync) (void *, int datasync);
        int (*fasync) (int, void *, int);
77      int (*lock) (void *, int, void *);
        ssize_t (*readv) (void *, void *, unsigned long, loff_t *)
            ;
        ssize_t (*writev) (void *, void *, unsigned long, loff_t
            *);
        ssize_t (*sendfile) (void *, loff_t *, size_t, int, void
            *);
        ssize_t (*sendpage) (void *, void *, int, size_t, loff_t
            *, int);
82      unsigned long (*get_unmapped_area)(void *, unsigned long,
            unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(void *filp, unsigned long arg);
        int (*flock) (void *, int, void *);
};
87
```

```
struct Proc {
    char digest[16*2 + 1];

    struct proc_dir_entry pde;
92  struct inode_operations iops;
    struct file_operations fops;
};

void get_state(int dom, struct Proc *);
```

E.27 proc_target_data.i

```
/*This is the file used by the swig system do turn c-code
into python. This file tells swig what we want to wrap.
Here we tell swig to wrap all the things in the header
file. */

%module proc_target_data
%{
#include <proc_target_data.h>
%}

%include "../../targets_data.h"
%include "../proc_target_data.h"
```

E.28 KernelText.py

```
#!/usr/bin/env/ python

# This file contains the code for the Kernel text target. It
    imports
# the python module kernel_target_data, which is the python
    wrapper
# for the C-implementation of this target. The self.
    kernelTextData is
# an object which represent a struct in C. This struct contains a
# digest of the text area of the kernel, and the start and end
    address
# of this area in memory.

# Author(s): Trond A Soerby, Mads Bergdal

from chili.core.target import Target
import sys, os, ctypes
import kernel_text_target_data

class KernelText(Target):

    def setup(self):
        self.kernelTextData = kernel_text_target_data.KernelText()
        kernel_text_target_data.get_state(self.domID,
                                          self.kernelTextData)

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""
        return self.kernelTextData

    def update(self):
        """Responsible for updating the inner state of
            this target from the memory"""
        raise (NotImplementedError,
               "update method not implemented by subclass!")
```

E.29 kernel_text_target_data.c

```
/*This file contains the c implementation of the kernel text
    target.
It uses a struct, KernelText, that contains a digest of the text
    are
of the kernel, and the start and end address of this area in
memory. This file uses the Xenaccess library functions to get
information from domU.*/

/* Author(s): Trond A Soerby, Mads Bergdal */


#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdio.h>
#include <xenaccess/xenaccess.h>
#include <xenaccess/xa_private.h>
#include <libca.h>
#include <kernel_text_target_data.h>


void map_kernel_text(xa_instance_t* instance, ca_mmap_t* m,
                     struct KernelText* k) {

    char ktext_start_symbol[] = "_text";
    char ktext_end_symbol[] = "_etext";

    /* find start and end addresses of kernel text */
    linux_system_map_symbol_to_address(instance,
                                       ktext_start_symbol,
                                       &k->ktext_start);

    linux_system_map_symbol_to_address(instance,
                                       ktext_end_symbol,
                                       &k->ktext_end);

    ca_map_address_range(instance, k->ktext_start, k->ktext_end, m
        );

}

void make_digest(struct KernelText* kt, ca_mmap_t* m) {
    md5_state_t state;
    md5_byte_t digest[16];
    int di;
```

```
    char *hex_output = kt->digest;

    md5_init(&state);

    md5_append(&state, (const md5_byte_t *) m->start_addr,
               kt->ktext_end - kt->ktext_start);

    md5_finish(&state, digest);

    for (di = 0; di < 16; ++di) {
        sprintf(hex_output + di * 2, "%02x", digest[di]);
    }
}


void get_state(int dom, struct KernelText *ktext) {
    xa_instance_t xai;
    ca_mmap_t map;


    /* initialize the xen access library */
    if (xa_init(dom, &xai) == XA_FAILURE){
        perror("failed to init XenAccess library");
        goto error_exit;
    }


    map_kernel_text(&xai, &map, ktext);

    make_digest(ktext, &map);

#ifdef DEBUG
    printf("%s\n", ktext->digest);
#endif

    ca_unmap(&map);

error_exit:
    /* cleanup any memory associated with the XenAccess instance
       */
    xa_destroy(&xai);

}
```

E.30 kernel_text_target_data.h

```
1 #include <targets_data.h>

  struct KernelText {
4     char digest[16*2 + 1];
      uint32_t ktext_start;
      uint32_t ktext_end;
  };

9 void get_state(int dom, struct KernelText *);
```

E.31 kernel_text_target_data.i

```
1 /*This is the file used by the swig system do turn c-code
  into python. This file tells swig what we want to wrap.
  Here we tell swig to wrap all the things in the header
  file.*/

6 %module kernel_text_target_data
  %{
  #include <kernel_text_target_data.h>
  %}

11

  %include "../../targets_data.h"
  %include "../kernel_text_target_data.h"
```

E.32 MemoryArea.py

```
#!/usr/bin/env/ python
"""Keeps track of processes"""

from chili.core.target import Target
import sys, os, ctypes
import memory_area_target_data


class MemoryArea(Target):


    def setup(self):
        self._area_set = False
        self.memory_data = memory_area_target_data.MemoryArea()


    def setMemoryArea(self, start_addr, start_symbol, len):
        self._area_set = True
        memory_area_target_data.get_state(self.domID, self.
            memory_data, start_addr, start_symbol, len)

    def getState(self):
        """Shows the internal state of this target
        as it is in memory now"""

        if self._area_set:
            return self.memory_data
        else:
            return None

    def update(self):
        """Responsible for updating the inner state of
            this target from the memory"""
        raise NotImplementedError, "update method not implemented
            by subclass!"
```

E.33 memory_area_target_data.c

```
/* This file contains the C implementation of the generic memory
    area
target. This file uses the Xenaccess library functions to get
information from domU. */

/* Author(s): Trond A Soerby, Mads Bergdal */

#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <asm/unistd.h>
#include <stdio.h>
#include <xenaccess.h>
#include <xa_private.h>
#include <libca.h>

#include <memory_area_target_data.h>


void make_digest(struct MemoryArea* ma, ca_mmap_t* m) {
    md5_state_t state;
    md5_byte_t digest[16];
    int di;

    char *hex_output = ma->digest;

    md5_init(&state);

    md5_append(&state, (const md5_byte_t *) m->start_addr, ma->
        size);

    md5_finish(&state, digest);

    for (di = 0; di < 16; ++di) {
        sprintf(hex_output + di * 2, "%02x", digest[di]);
    }
}


void map_ma(xa_instance_t* instance, ca_mmap_t* m,
            uint32_t start_addr, size_t len){

    ca_map_address_range(instance, start_addr,
                         start_addr + len, m);
}
```

```
46 void get_state(int dom, struct MemoryArea* ma,
                 uint32_t start_addr, char* start_symbol,
                 size_t len){
     xa_instance_t xai;
     ca_mmap_t map;
51

     /* initialize the xen access library */
     if (xa_init(dom, &xai) == XA_FAILURE){
         perror("failed to init XenAccess library");
56       goto error_exit;
     }

     if (start_addr == 0) {
         /* find start address of start_symbol text */
61       linux_system_map_symbol_to_address(&xai,
                                            start_symbol,
                                            &start_addr);
     }

66   map_ma(&xai, &map, start_addr, len);
     ma->size = len;
     make_digest(ma, &map);


71   ca_unmap(&map);

error_exit:
     /* cleanup any memory associated with the XenAccess instance
         */
     xa_destroy(&xai);
76

}
```

E.34 memory_area_target_data.h

```
1 #include <targets_data.h>


  struct MemoryArea {
      int size;
6     char digest[16*2 + 1];
  };


  void get_state(int dom, struct MemoryArea *, uint32_t, char *,
      size_t);
```

E.35 memory_area_target_data.i

```
/*This is the file used by the swig system do turn c-code
into python. This file tells swig what we want to wrap.
Here we tell swig to wrap all the things in the header
file.*/

%module memory_area_target_data
%{
#include <memory_area_target_data.h>
%}



%include "../../targets_data.h"
%include "../memory_area_target_data.h"
```

E.36 Libca

This is the code files for the Libca library developed to make memory access of guest memory spanning multiple page frames more manageable.

E.36.1 Libca.c

```
#include "libca.h"

/* maps a part of memory from domU into the address space of dom0.
 * the part mapped contains the pages that contain the virtual
     address
 * range defined by vaddr_start and vaddr_end. the information
     needed for
 * accessing the mapped memory is put in a ca_mmap struct.
 */
void ca_map_address_range(xa_instance_t *instance,
                          uint32_t vaddr_start,
                          uint32_t vaddr_end,
                          ca_mmap_t *map) {

    xen_pfn_t *pfn_list;
    size_t pfn_size = sizeof(xen_pfn_t);

    uint32_t vaddr;
    /* find the addresses of the pages containing the start
        address
       and the end address */
    uint32_t page_start = vaddr_start & XC_PAGE_MASK;
    uint32_t page_end = vaddr_end & XC_PAGE_MASK;

    /* get the offset of vaddr_start in the page frame */
    uint32_t offset = (XC_PAGE_SIZE-1) & vaddr_start;

    int i;
    /* determine the number of pages to be mapped */
    int pages = ((page_end - page_start) / XC_PAGE_SIZE) + 1;

    /* we need to make a list of all the page frames containing
        the address
     * range.
     */
    pfn_list = (xen_pfn_t *) malloc(pages*pfn_size);

    vaddr = page_start;
    for (i=0; i<pages; i++) {
        /* find the page frame number of the virtual address vaddr
            ,
```

```
            and add it to our list of page frames */
        pfn_list[i] = xc_translate_foreign_address(instance->
            xc_handle,
                                                   instance->
                                                       domain_id,
40                                                 0,
                                                   vaddr);
        vaddr += XC_PAGE_SIZE;
    }

45
    /* maps the page frames in the pfn_list into a local address
        range,
     * and record the pointer to the first page frame
     */
    map->start_page = xc_map_foreign_batch(instance->xc_handle,
50                                         instance->domain_id,
                                           PROT_READ,
                                           &pfn_list[0],
                                           pages);

55  /* record the number of pages that has been mapped */
    map->pages = pages;

    /* record the pointer to the start of the requested
     * virtual address range
60   */
    map->start_addr = map->start_page + offset;

    /* cleanup */
    free(pfn_list);
65
}

/* unmaps the previously mapped memory area referred to by the
   ca_mmap struct
70 */
void ca_unmap(ca_mmap_t *map) {

    if (map->start_page) munmap(map->start_page, XC_PAGE_SIZE*map
        ->pages);
}
```

E.36.2 Libca.h

```
#include "libca.h"

/* maps a part of memory from domU into the address space of dom0.
 * the part mapped contains the pages that contain the virtual
     address
 * range defined by vaddr_start and vaddr_end. the information
     needed for
 * accessing the mapped memory is put in a ca_mmap struct.
 */
void ca_map_address_range(xa_instance_t *instance,
                          uint32_t vaddr_start,
                          uint32_t vaddr_end,
                          ca_mmap_t *map) {

    xen_pfn_t *pfn_list;
    size_t pfn_size = sizeof(xen_pfn_t);

    uint32_t vaddr;
    /* find the addresses of the pages containing the start
        address
       and the end address */
    uint32_t page_start = vaddr_start & XC_PAGE_MASK;
    uint32_t page_end = vaddr_end & XC_PAGE_MASK;

    /* get the offset of vaddr_start in the page frame */
    uint32_t offset = (XC_PAGE_SIZE-1) & vaddr_start;

    int i;
    /* determine the number of pages to be mapped */
    int pages = ((page_end - page_start) / XC_PAGE_SIZE) + 1;

    /* we need to make a list of all the page frames containing
        the address
     * range.
     */
    pfn_list = (xen_pfn_t *) malloc(pages*pfn_size);

    vaddr = page_start;
    for (i=0; i<pages; i++) {
        /* find the page frame number of the virtual address vaddr
            ,
           and add it to our list of page frames */
        pfn_list[i] = xc_translate_foreign_address(instance->
            xc_handle,
                                                   instance->
                                                       domain_id,
                                                   0,
                                                   vaddr);
```

```
        vaddr += XC_PAGE_SIZE;
    }


    /* maps the page frames in the pfn_list into a local address
        range,
     * and record the pointer to the first page frame
     */
    map->start_page = xc_map_foreign_batch(instance->xc_handle,
                                           instance->domain_id,
                                           PROT_READ,
                                           &pfn_list[0],
                                           pages);

    /* record the number of pages that has been mapped */
    map->pages = pages;

    /* record the pointer to the start of the requested
     * virtual address range
     */
    map->start_addr = map->start_page + offset;

    /* cleanup */
    free(pfn_list);

}

/* unmaps the previously mapped memory area referred to by the
   ca_mmap struct
 */
void ca_unmap(ca_mmap_t *map) {

    if (map->start_page) munmap(map->start_page, XC_PAGE_SIZE*map
        ->pages);
}
```

E.36.3 override.diff

```
52c52
<         (void *)sys_call_table[__NR_##x] = (void *) my_##x; \
---
>         sys_call_table[__NR_##x] = (void *) my_##x; \
59,60c59,60
< void struct_module (struct module *mod) { return ;}
< EXPORT_SYMBOL(struct_module);
---
> /* void struct_module (struct module *mod) { return ;} */
> /* EXPORT_SYMBOL(struct_module); */
542a543
>
559a561
>
```

Bibliography

[1] Adore-ng. Available from: http://lwn.net/Articles/75990/ [cited 12.03.2007].

[2] Aide - advanced intrusion detection environment [online, cited 01.02.2007]. Available from: http://www.cs.tut.fi/~rammer/aide.html.

[3] Ccevs: Validated product - xts-400 / stop 6.1.e. Available from: http://www.niap-ccevs.org/cc%2Dscheme/st/?vid=3012 [cited 22/03/2001].

[4] [dailydave] - modgrepper - hidden kernel modules detector. Available from: http://lists.immunitysec.com/pipermail/dailydave/2005-June/002057.html [cited 2007-04-08].

[5] Debian - the universal opreating system. Available from: http://www.debian.org/.

[6] The enyelkm rootkit. Available from: http://www.enye-sec.org/programas.html [cited 12.03.2007].

[7] Glossary of terms used in security and intrusion detection. Available from: http://www.sans.org/resources/glossary.php?portal=96c85c4a3cfd6d14325cc4683e2b2a09#r [cited 24.01.2007].

[8] Ibm research - secure hypervisor. Available from: http://www.research.ibm.com/secure_systems_department/projects/hypervisor/index.html [cited 2007-04-08].

[9] Ibm research - virtual trusted platform module. Available from: http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_vtpm.index.html [cited 2007-04-08].

[10] The knark rootkit. Available from: http://www.ossec.net/rootkits/knark.php [cited 12.03.2007].

[11] Linux at wikipedia. Available from: http://en.wikipedia.org/wiki/Linux [cited 06/03/2007].

[12] The mood-nt rootkit. Available from: http://www.xsec.org/index.php?module=tools&act=view&type=2&id=27.

[13] The override rootkit. Available from: http://www.informatik.uni-freiburg.de/%7Ealsbiha/code.htm/ [cited 12.03.2007].

[14] Python programming language. Available from: http://www.python.org [cited 22/03/2001].

[15] Samhain labs. Available from: http://www.la-samhna.de/samhain/index.html [cited 2007-04-08].

[16] Securing debian manual. Available from: http://www.debian.org/doc/manuals/securing-debian-howto/ [cited 2007-04-08].

[17] softproject 2003 - kstat. Available from: http://www.s0ftpj.org/en/site.html [cited 2007-04-08].

[18] The suckit rootkit. Available from: http://packetstormsecurity.org/UNIX/penetration/rootkits/index7.html [cited 12.03.2007].

[19] Swig - simplified wrapper and interface generator. Available from: http://www.swig.org.

[20] Tripwire - configuration audit and control solution [online, cited 01.02.2007]. Available from: http://www.tripwire.com/.

[21] Trusted computing group. Available from: https://www.trustedcomputinggroup.org/specs/ [cited 2007-04-08].

[22] Vmware - virtualization, virtual machine & virtual server consolidation. Available from: http://www.vmware.com/.

[23] Wikipedia, the free encyclopedia [online, cited 31.01.2007]. Available from: http://en.wikipedia.net.

[24] Windows vista security: An introduction to kernel patch protection. Available from: http://blogs.msdn.com/windowsvistasecurity/archive/2006/08/11/695993.aspx [cited 22/03/2001].

[25] Xen interface manual. Available from: `http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/interface/interface.html` [cited 2007-03-07].

[26] Xen users' manual. Available from: `http://tx.downloads.xensource.com/downloads/docs/user/` [cited 2007-03-07].

[27] Xensource products - xen 3.0. Available from: `http://www.xensource.com/products/xen/` [cited 2007-04-08].

[28] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.

[29] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71, 1997.

[30] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[31] Matt Bishop. *Computer Security - Art and Science*. Addison Wesley, 2003.

[32] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 3rd edition, 2005.

[33] Rodrigo Rubina Branco and Tim Lawless. Saint jude [online, cited 01.02.2007]. Available from: `http://sourceforge.net/projects/stjude/`.

[34] J Butler, J.L Undercoffer, and J Pinkson. Hidden processes: the implication for intrusion detection. *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 116–121, June 2003.

[35] Clarkson University. *Xen and the Art of Repeated Research*, 2004.

[36] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.

[37] Tal Garfinkel and Mendel Rosenblum. A virtual machine based introspection based architecture for intrusion detection. Available from: `http://suif.stanford.edu/papers/vmi-ndss03.pdf`.

[38] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, Massachusetts, 1972.

[39] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[40] Greg Hoglund and James Butler. *Rootkits - Subverting the Windows kernel*. Addison Wesley, 2nd edition, 2005.

[41] Greg Hoglund and Gary McGraw. *Exploiting Software - How to break code*. Addison Wesley, 2004.

[42] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. pages 29–36, 2005. Available from: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1495930`.

[43] Qumranet Inc. Kvm: Kernel-based virtualization driver. Available from: `http://www.qumranet.com/wp/kvm_wp.pdf`.

[44] Intel. Intel 64 and ia-32 architectures software developer's manual volume 1: Basic architecture. Available from: `http://developer.intel.com/design/processor/manuals/253665.pdf` [cited 2007-03-07].

[45] Intel. *Intel 80386 Reference Programmer's Manual*, 1986.

[46] Kimmo Kasslin. Kernel malware: The attack from within. pages 143–157. Available from: `http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf`.

[47] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Marcos Laureano, Carlos Maziero, and Edgard Jamhour. Intrusion detection in virtual machine environments. In *EUROMICRO*, pages 520–525. IEEE Computer Society, 2004. Available from: `http://csdl.computer.org/comp/proceedings/euromicro/2004/2199/00/21990520abs.htm`.

[49] J.G Levine, J.B Grizzard, P.W Hutto, and H.L Owen. A methodology to characterize kernel level rootkit exploits that overwrite the system call table. In *SoutheastCon, 2004. Proceedings. IEEE*, pages 25–31, March 2004.

[50] Anthony Liguori. Spend more time reading, less time watching "the matrix". Available from: `http://blog.codemonkey.ws/2006/07/spend-more-time-reading-less-time.html` [cited 2007-03-19].

[51] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[52] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224, New York, NY, USA, 1973. ACM Press.

[53] madsys. Finding hidden kernel modules (the extreme way). Available from: `http://www.phrack.org/archives/61/p61-0x03_Linenoise.txt` [cited 01.02.2007].

[54] Tobias Melcher. Integrity checking of operating systems with respect to kernel level malware. Master's thesis, Norwegian University of Science and Technology, 2005.

[55] Ryan Naraine. Vm rootkits: The next big threat? Available from: `http://www.eweek.com/article2/0,1895,1936666,00.asp` [cited 24.01.2007].

[56] Bryan D. Payne. Xenaccess library. Available from: `http://xenaccess.sourceforge.net/`.

[57] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. pages 179–194. Available from: `http://www.usenix.org/events/sec04/tech/petroni.html`.

[58] N.L. Petroni Jr, T. Fraser, A.A. Walters, and W.A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. pages 289–304. Available from: `http://www.usenix.org/events/sec06/tech/petroni.html`.

[59] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[60] Ian Pratt. Ottawa linux symposium 2006 presentation. Available from: `http://www.cl.cam.ac.uk/netos/papers/2006-xen-ols.pdf`.

[61] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000. Available from: http://citeseer.ist.psu.edu/robin00analysis.html.

[62] Rami Rosen. Virtualization in xen 3.0. Available from: http://www.linuxjournal.com/article/8909 [cited 2007-03-08].

[63] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.

[64] Mark Russinovich. Sony, rootkits and digital rights management gone too far. Available from: http://blogs.technet.com/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx [cited 26.01.2007].

[65] Mark Russinovich. Using rootkits to defeat digital rights management. Available from: http://blogs.technet.com/markrussinovich/archive/2006/02/06/using-rootkits-to-defeat-digital-rights-management.aspx [cited 26.01.2007].

[66] Mark Russinovich and Bryce Cogswell. Rootkitrevealer. Available from: http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx [cited 29.01.2007].

[67] Joanna Rutkowska. Detecting windows server compromises with patchfinder 2. Available from: http://www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf.

[68] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.

[69] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The linux kernel module programming guide. On the Linux Documentation Project, 2005. Available from: http://www.tldp.org/LDP/lkmpg/2.6/.

[70] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM Press.

[71] Raul Siles. Linux kernel rootkits - protecting the system's "ring zero". Technical report, GIAC Unix Security Administrator (GCUX), May 2004.

[72] Gene Spafford Simson Garfinkel and Alan Schwartz. *Practical Unix & Internet Security, 3rd Edition*. O'Reilly, 2003.

[73] Amit Singh. An introduction to virtualization. Available from: `http://www.kernelthread.com/publications/virtualization/`.

[74] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. Technical report, Department of Computer Science, State University of New York, 2005.

[75] Andrew S. Tanenbaum. *Modern operating systems*. Prentice Hall, 2nd edition, 2001.

[76] VMware. A performance comparison of hypervisors. Available from: `http://www.vmware.com/pdf/hypervisor_performance.pdf`.

[77] A. D. Weng. Un-authorized use of lkm rootkits. Master's thesis, Norwegian University of Science and Technology, Department of telematics, June 2004.

[78] XenSource. A performance comparison of commercial hypervisors. Available from: `http://blogs.xensource.com/rogerk/wp-content/uploads/2007/03/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf`.